
The J2EE™ Tutorial for the Sun™ ONE Platform

Eric Armstrong
Stephanie Bodoff
Maydene Fisher
Dale Green
Kim Haase

February 13, 2003

Copyright © 2003 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. All rights reserved. U.S. Government Rights - Commercial software. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements.

This distribution may include materials developed by third parties.

Sun, Sun Microsystems, the Sun logo, Java, J2EE, JavaServer Pages, Enterprise JavaBeans, Java Naming and Directory Interface, EJB, JSP, J2EE, J2SE and the Java Coffee Cup logo are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Unless otherwise licensed, software code in all technical materials herein (including articles, Fads, samples) is provided under this License.

Products covered by and information contained in this service manual are controlled by U.S. Export Control laws and may be subject to the export or import laws in other countries. Nuclear, missile, chemical biological weapons or nuclear maritime end uses or end users, whether direct or indirect, are strictly prohibited. Export or reexport to countries subject to U.S. embargo or to entities identified on U.S. export exclusion lists, including, but not limited to, the denied persons and specially designated nationals lists is strictly prohibited.

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright © 2003 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, États-Unis. Tous droits réservés. Droits du gouvernement américain, utilisateurs gouvernementaux - logiciel commercial. Les utilisateurs gouvernementaux sont soumis au contrat de licence standard de Sun Microsystems, Inc., ainsi qu'aux dispositions en vigueur de la FAR [(Federal Acquisition Regulations) et des suppléments à celles-ci.

Cette distribution peut comprendre des composants développés par des tierces parties.

Sun, Sun Microsystems, le logo Sun, Java, JavaServer Pages, Enterprise JavaBeans, Java Naming and Directory Interface, EJB, JSP, J2EE, J2SE et le logo Java Coffee Cup sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux États-Unis et dans d'autres pays.

A moins qu'autrement autorisé, le code de logiciel en tous les matériaux techniques dans le présent (articles y compris, FAQs, échantillons) est fourni sous ce permis.

Les produits qui font l'objet de ce manuel d'entretien et les informations qu'il contient sont régis par la législation américaine en matière de contrôle des exportations et peuvent être soumis au droit d'autres pays dans le domaine des exportations et importations. Les utilisations finales, ou utilisateurs finaux, pour des armes nucléaires, des missiles, des armes biologiques et chimiques ou du nucléaire maritime, directement ou indirectement, sont strictement interdites. Les exportations ou réexportations vers des pays sous embargo des États-Unis, ou vers des entités figurant sur les listes d'exclusion d'exportation américaines, y compris, mais de manière non exclusive, la liste de personnes qui font objet d'un ordre de ne pas participer, d'une façon directe ou indirecte, aux exportations des produits ou des services qui sont régi par la législation américaine en matière de contrôle des exportations ("U.S. Commerce Department's Table of Denial Orders" et la liste de ressortissants spécifiquement désignés ("U.S. Treasury Department of Specially Designated Nationals and Blocked Persons"),, sont rigoureusement interdites.

LA DOCUMENTATION EST FOURNIE "EN L'ÉTAT" ET TOUTES AUTRES CONDITIONS, DÉCLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISÉE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE À LA QUALITÉ MARCHANDE, À L'APTITUDE À UNE UTILISATION PARTICULIÈRE OU À L'ABSENCE DE CONTREFAÇON.

Contents

	About This Tutorial.	vii
	Who Should Use This Tutorial	vii
	Prerequisites	viii
	How to Read This Tutorial	viii
	About the Examples	ix
	Further Information	x
	How to Print This Tutorial	xi
	Typographical Conventions	xi
Chapter 1:	Overview.	1
	Web Services Support	3
	Distributed Multitiered Applications	5
	J2EE Containers	11
	Packaging	13
	Development Roles	14
	J2EE APIs	17
Chapter 2:	Introduction to Web Applications.	25
	Web Application Life Cycle	26
	Web Modules	28
	Configuring Web Modules	30
	Deploying Web Modules	33
	Listing Deployed Web Modules	34
	Running Web Applications	34
	Updating Web Modules	35
	Undeploying Web Modules	36
	Duke's Bookstore	37
	Internationalizing and Localizing Web Applications	38
	Accessing Databases from Web Applications	39

	Further Information	45
Chapter 3:	Java Servlet Technology	47
	What is a Servlet?	47
	The Example Servlets	48
	Servlet Life Cycle	51
	Sharing Information	54
	Initializing a Servlet	58
	Writing Service Methods	59
	Filtering Requests and Responses	64
	Invoking Other Web Resources	72
	Accessing the Web Context	75
	Maintaining Client State	76
	Finalizing a Servlet	79
	Further Information	82
Chapter 4:	JavaServer Pages Technology	83
	What Is a JSP Page?	83
	The Example JSP Pages	86
	The Life Cycle of a JSP Page	88
	Initializing and Finalizing a JSP Page	91
	Creating Static Content	92
	Creating Dynamic Content	92
	Including Content in a JSP Page	98
	Transferring Control to Another Web Component	100
	Including an Applet	100
	JavaBeans Components in JSP Pages	103
	Extending the JSP Language	111
	Further Information	112
Chapter 5:	Custom Tags in JSP Pages	113
	What Is a Custom Tag?	114
	The Example JSP Pages	114
	Using Tags	118
	Defining Tags	122
	Examples	137

Chapter 6:	JavaServer Pages Standard Tag Library	149
	The Example JSP Pages	150
	Using JSTL	151
	Expression Language Support	153
	Core Tags	159
	XML Tags	165
	Internationalization Tags	169
	SQL Tags	172
	Further Information	176
Chapter 7:	Understanding XML	177
	Introduction to XML	177
	XML and Related Specs: Digesting the Alphabet Soup	187
	Generating XML Data	199
	Designing an XML Data Structure	233
Chapter 8:	Introduction to Web Services	239
	The Role of XML and the Java Platform	240
	Overview of the Java APIs for XML	241
	JAXP	242
	JAX-RPC	250
	JAXM	256
	JAXR	263
	Sample Scenario	265
Chapter 9:	Building Web Services With JAX-RPC	269
	Types Supported By JAX-RPC	270
	Creating a Web Service with JAX-RPC	272
	Creating Web Service Clients with JAX-RPC	276
	Further Information	286
Chapter 10:	Web Services Messaging with JAXM	289
	The Structure of the JAXM API	290
	Overview of JAXM	291
	Tutorial	302
	Code Examples	323
	Conclusion	345

Further Information	346
Chapter 11: Publishing and Discovering Web Services with JAXR	347
Overview of JAXR	347
Implementing a JAXR Client	351
Running the Client Examples	371
Further Information	382
Chapter 12: The Coffee Break Application	383
Coffee Break Overview	383
JAX-RPC Distributor Service	385
JAXM Distributor Service	393
Coffee Break Server	401
Deploying and Running the Application	415
Appendix A: Java Encoding Schemes	425
Further Information	426
Appendix B: HTTP Overview	427
HTTP Requests	428
HTTP Responses	428
Glossary	429
About the Authors	457
Index	459

About This Tutorial

THIS tutorial is a beginner's guide to developing J2EE applications for the Sun™ Open Networking Environment (Sun ONE) platform. Here we cover all the things you need to know to make the best use of this tutorial.

Who Should Use This Tutorial

This tutorial is intended for programmers interested in developing J2EE applications using the Sun ONE platform. Specifically, it uses the Sun ONE Studio 4, Enterprise Edition for Java to develop J2EE applications and deploy them on Sun ONE Application Server 7.

Note: This release of the tutorial includes information on Web applications and Web services. A future release will add coverage of Enterprise JavaBeans technology and J2EE platform services such as transactions, security, resources, and connectors.

This tutorial is not a comprehensive introduction to Sun ONE platform software. It focuses on teaching the concepts of J2EE technologies through an extensive set of examples and provides basic instructions in how to use the software so that you can develop, configure, build, deploy, and run the examples. To learn how to use the software, see the resources listed in Further Information (page x).

Prerequisites

To understand this tutorial you will need a good knowledge of the Java programming language, SQL, and relational database concepts. The topics listed in Table P–1 *The Java™ Tutorial* are particularly relevant:

Table P–1 Relevant Topics in *The Java™ Tutorial*

Topic	Web Page
JDBC™	http://java.sun.com/docs/books/tutorial/jdbc
Threads	http://java.sun.com/docs/books/tutorial/essential/threads
JavaBeans™	http://java.sun.com/docs/books/tutorial/javabeans

How to Read This Tutorial

This tutorial is organized into the following sections:

- Introduction

These chapters introduce the J2EE platform and the fundamentals of Web applications. They are prerequisites for the rest of the tutorial.
- Overview
- Web Applications
- Web Technology

These chapters cover the technologies used in developing presentation-oriented Web applications.

 - Java Servlets
 - JavaServer Pages
 - JSP custom tags
 - The JSP Standard Tag Library (JSTL)
- Java Web Services Technology

These chapters cover the technologies used in developing service-oriented Web applications.

- Defining RPC-oriented Web services with the Java API for XML-based RPC (JAX-RPC).
 - Defining message-oriented Web services with the Java API for XML Messaging (JAXM) and Soap with Attachments API for Java (SAAJ).
 - Publishing and discovering RPC-oriented Web services. The Java API for XML Registries (JAXR)
 - Case Study
- The Coffee Break Application chapter describes an application that ties together most of the APIs discussed in this tutorial.

About the Examples

If you are viewing this online, you need to download *The J2EE™ Tutorial for the Sun ONE Platform* from:

<http://java.sun.com/j2ee/1.3/download.html#tutorial>

Once you have unzipped the tutorial bundle, the example source code is in the `<INSTALL>/j2eetutorial/examples` directory, with subdirectories for each of the technologies. Note that `<INSTALL>` is the directory in which you unzipped the bundle.

To build the examples you need a copy of Sun ONE Studio 4 update 1, Enterprise Edition for Java. You download a free trial copy of this edition of Studio from:

<http://www.sun.com/software/sundev/jde/index.html>

To deploy and run the examples, you need a copy of Sun ONE Application Server 7, Platform Edition or Standard Edition. Sun ONE Application Server 7, Platform Edition is free for development and deployment, but does not allow remote management. Sun ONE Application Server 7, Standard Edition is free for development and evaluation and supports remote management. You can download either edition of the application server from:

http://www.sun.com/software/products/appsrvr/home_appsrvr.html

To deploy the examples from Sun ONE Studio to Application Server 7, you also need to install the Sun ONE Application Server 7 Plugin into Studio. You can get the Plugin from the Studio Update Center or the Application Server 7 installa-

tion. *Sun™ ONE Studio 4, Enterprise Edition for Java™ with Application Server 7 Tutorial* contains information on how to install the Plugin.

Further Information

Each chapter in the tutorial contains pointers to additional resources on the technologies described in the chapter.

For basic information on how to use Sun ONE Studio with Sun ONE Application Server 7 see *Sun™ ONE Studio 4, Enterprise Edition for Java™ with Application Server 7 Tutorial* located at `<S1AS7_HOME>/docs/studio-tutorial/index.html`.

For further information on the Sun ONE platform components see the following resources.

Sun ONE Application Server 7

- *Getting Started with Sun™ ONE Application Server 7* located at `<S1AS7_HOME>/docs/getting-started/index.html`
- For complete documentation for Sun ONE Application Server 7, go to `http://docs.sun.com/db/coll/s1_asse_en`.

Sun ONE Studio 4, Enterprise Edition for Java

- *Sun™ ONE Studio 4, Enterprise Edition for Java™ Tutorial* `http://forte.sun.com/ffj/documentation/s1s41/s1seetut.pdf`
- Sun™ ONE Studio 4, Enterprise Edition for Java™ update 1 online help
- *Building J2EE Applications* `http://forte.sun.com/ffj/documentation/s1s41/j2eeapps.pdf`
- *Building Web Components* `http://forte.sun.com/ffj/documentation/s1s41/webcomp.pdf`
- *Building Web Services* `http://forte.sun.com/ffj/documentation/s1s41/websrvcs.pdf`
- For complete documentation for Sun ONE Studio 4, go to `http://www.sun.com/software/sundev/jde/documentation/index.html`.

How to Print This Tutorial

To print this tutorial, follow these steps:

1. Ensure that Adobe Acrobat Reader is installed on your system.
2. Open the PDF version of this book.
3. Click the printer icon in Adobe Acrobat Reader.

Typographical Conventions

Table P–2 lists the typographical conventions used in this tutorial.

Table P–2 Typographical Conventions

Font Style	Uses
<i>italic</i>	Emphasis, titles, first occurrence of terms
monospace	URLs, code examples, file names, command names, programming language keywords
<i>italic monospace</i>	Variable file names

Menu selections indicated with the right-arrow character →, for example, First→Second, should be interpreted as: select the First menu, then choose Second from the First submenu.

Overview

Monica Pawlan

TODAY, more and more developers want to write distributed transactional applications for the enterprise and leverage the speed, security, and reliability of server-side technology. If you are already working in this area, you know that in today's fast-moving and demanding world of e-commerce and information technology, enterprise applications have to be designed, built, and produced for less money, with greater speed, and with fewer resources than ever before.

To reduce costs and fast-track application design and development, Java™ 2 Platform, Enterprise Edition (J2EE™) provides a component-based approach to the design, development, assembly, and deployment of enterprise applications. The J2EE platform offers a multitiered distributed application model, reusable components, a unified security model, flexible transaction control, and Web services support through integrated data interchange on eXtensible Markup Language (XML)-based open standards and protocols.

Not only can you deliver innovative business solutions to market faster than ever, but your platform-independent J2EE component-based solutions are not tied to the products and application programming interfaces (APIs) of any one vendor. Vendors and customers enjoy the freedom to choose the products and components that best meet their business and technological requirements.

This tutorial takes an examples-based approach to describing the features and functionalities available in J2EE Software Development Kit (SDK) version 1.4 for developing enterprise applications. Whether you are a new or an experienced developer, you should find the examples and accompanying text a valuable and accessible knowledge base for creating your own solutions.

If you are new to J2EE enterprise application development, this chapter is a good place to start. Here you will learn development basics, be introduced to the J2EE architecture and APIs, become acquainted with important terms and concepts, and find out how to approach J2EE application programming, assembly, and deployment.

Web Services Support

Web services are Web-based enterprise applications that use open-XML-based standards and transport protocols to exchange data with calling clients. The J2EE platform provides the XML APIs and tools you need to quickly design, develop, test, and deploy Web services and clients that fully interoperate with other Web services and clients running on Java-based or non-Java-based platforms.

It is easy to write Web services and clients with the J2EE XML APIs. All you do is pass parameter data to the method calls and process the data returned, or for document-oriented web services, send documents containing the service data back and forth. No low-level programming is needed because the XML API implementations do the work of translating the application data to and from an XML-based data stream that is sent over the standardized XML-based transport protocols. These XML-based standards and protocols are introduced in the next sections.

The translation of data to a standardized XML-based data stream is what makes Web services and clients written with the J2EE XML APIs fully interoperable. This does not necessarily mean the data being transported includes XML tags because the transported data can itself be plain text, XML data, or any kind of binary data such as audio, video, maps, program files, CAD documents or the like. The next section, Extensible Markup Language (XML) (page 3), introduces XML and explains how parties doing business can use XML tags and schemas to exchange data in a meaningful way.

Extensible Markup Language (XML)

Extensible Markup Language (XML) is a cross-platform, extensible, and text-based standard for representing data. When XML data is exchanged between parties, the parties are free to create their own tags to describe the data, set up schemas to specify which tags can be used in a particular kind of XML document, and use XML style sheets to manage the display and handling of the data.

For example, a Web service can use XML and a schema to produce price lists, and companies that receive the price lists and schema can have their own style sheets to handle the data in a way that best suits their needs.

- One company might put the XML pricing information through a program to translate the XML to HTML so it can post the price lists to its Intranet.
- A partner company might put the XML pricing information through a tool to create a marketing presentation.
- Another company might read the XML pricing information into an application for processing.

HTTP-SOAP Transport Protocol

Client requests and Web service responses are transmitted as Simple Object Access Protocol (SOAP) messages over HTTP to enable a completely interoperable exchange between clients and Web services all running on different platforms and at various locations on the Internet. HTTP is a familiar request and response standard for sending messages over the Internet, and SOAP is an XML-based protocol that follows the HTTP request and response model.

The SOAP portion of a transported message handles the following:

- Defines an XML-based envelope to describe what is in the message and how to process the message.
- Includes XML-based encoding rules to express instances of application-defined data types within the message.
- Defines an XML-based convention for representing the request to the remote service and the resulting response.

WSDL Standard Format

The Web Services Description Language (WSDL) is a standardized XML format for describing network services. The description includes the name of the service, the location of the service, and how to communicate with the service. WSDLs can be stored in UDDI registries and/or published on the Web. The J2EE platform provides a tool for generating the WSDL for a Web service that uses remote procedure calls to communicate with clients.

UDDI and ebXML Standard Formats

Other XML-based standards such as Universal Description, Discovery, and Integration (UDDI) and ebXML make it possible for businesses to publish information on the Internet about their products and Web services where the information can be readily and globally accessed by clients who want to do business.

Distributed Multitiered Applications

The J2EE platform uses a multitiered distributed application model for both enterprise applications. Application logic is divided into components according to function, and the various application components that make up a J2EE application are installed on different machines depending on the tier in the multitiered J2EE environment to which the application component belongs. Figure 1–1 shows two multitiered J2EE applications divided into the tiers described in the following list. The J2EE application parts shown in Figure 1–1 are presented in J2EE Components (page 6).

- Client-tier components run on the client machine.
- Web-tier components run on the J2EE server.
- Business-tier components run on the J2EE server.
- Enterprise information system (EIS)-tier software runs on the EIS server.

Although a J2EE application can consist of the three or four tiers shown in Figure 1–1, J2EE multitiered applications are generally considered to be three-tiered applications because they are distributed over three different locations: client machines, the J2EE server machine, and the database or legacy machines at the back end. Three-tiered applications that run in this way extend the standard two-tiered client and server model by placing a multithreaded application server between the client application and back-end storage.

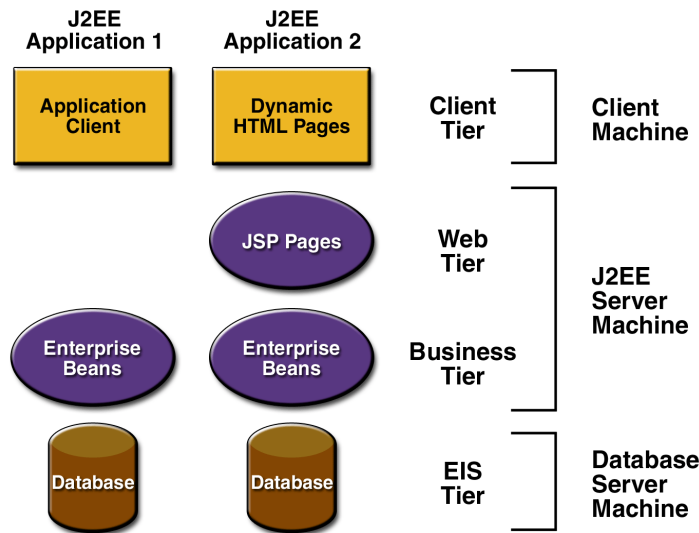


Figure 1–1 Multitiered Applications

J2EE Components

J2EE applications are made up of components. A *J2EE component* is a self-contained functional software unit that is assembled into a J2EE application with its related classes and files and that communicates with other components. The J2EE specification defines the following J2EE components:

- Application clients and applets are components that run on the client.
- Java Servlet and JavaServer Pages™ (JSP™) technology components are Web components that run on the server.
- Enterprise JavaBeans™ (EJB™) components (enterprise beans) are business components that run on the server.

J2EE components are written in the Java programming language and are compiled in the same way as any program in the language. The difference between J2EE components and “standard” Java classes is that J2EE components are assembled into a J2EE application, verified to be well formed and in compliance with the J2EE specification, and deployed to production, where they are run and managed by the J2EE server.

J2EE Clients

A J2EE client can be a Web client or an application client.

Web Clients

A Web client consists of two parts: dynamic Web pages containing various types of markup language (HTML, XML, and so on), which are generated by Web components running in the Web tier, and a Web browser, which renders the pages received from the server.

A Web client is sometimes called a *thin client*. Thin clients usually do not do things like query databases, execute complex business rules, or connect to legacy applications. When you use a thin client, heavyweight operations like these are off-loaded to enterprise beans executing on the J2EE server where they can leverage the security, speed, services, and reliability of J2EE server-side technologies.

Applets

A Web page received from the Web tier can include an embedded applet. An applet is a small client application written in the Java programming language that executes in the Java virtual machine installed in the Web browser. However, client systems will likely need the Java Plug-in and possibly a security policy file in order for the applet to successfully execute in the Web browser.

Web components are the preferred API for creating a Web client program because no plug-ins or security policy files are needed on the client systems. Also, Web components enable cleaner and more modular application design because they provide a way to separate applications programming from Web page design. Personnel involved in Web page design thus do not need to understand Java programming language syntax to do their jobs.

Application Clients

A J2EE application client runs on a client machine and provides a way for users to handle tasks that require a richer user interface than can be provided by a markup language. It typically has a graphical user interface (GUI) created from Swing or Abstract Window Toolkit (AWT) APIs, but a command-line interface is certainly possible.

Application clients directly access enterprise beans running in the business tier. However, if application requirements warrant it, a J2EE application client can open an HTTP connection to establish communication with a servlet running in the Web tier.

JavaBeans™ Component Architecture

The server and client tiers might also include components based on the JavaBeans component architecture (JavaBeans component) to manage the data flow between an application client or applet and components running on the J2EE server or between server components and a database. JavaBeans components are not considered J2EE components by the J2EE specification.

JavaBeans components have instance variables and `get` and `set` methods for accessing the data in the instance variables. JavaBeans components used in this way are typically simple in design and implementation, but should conform to the naming and design conventions outlined in the JavaBeans component architecture.

J2EE Server Communications

Figure 1–2 shows the various elements that can make up the client tier. The client communicates with the business tier running on the J2EE server either directly or, as in the case of a client running in a browser, by going through JSP pages or servlets running in the Web tier.

Your J2EE application uses a thin browser-based client or thick application client. In deciding which one to use, you should be aware of the trade-offs between keeping functionality on the client and close to the user (thick client) and off-loading as much functionality as possible to the server (thin client). The more functionality you off-load to the server, the easier it is to distribute, deploy, and manage the application; however, keeping more functionality on the client can make for a better perceived user experience.

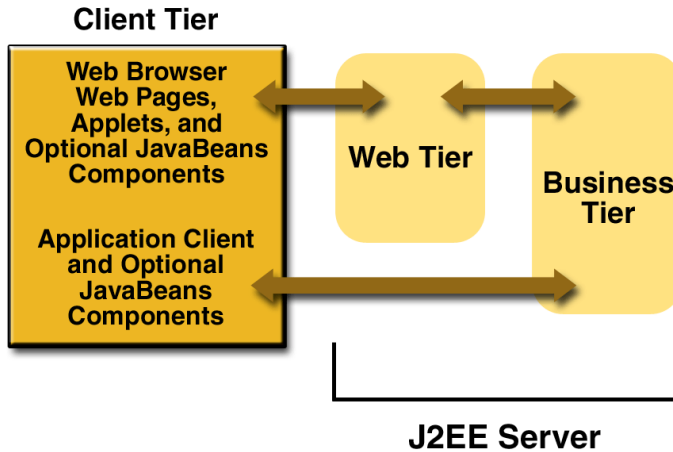


Figure 1–2 Server Communications

Web Components

J2EE Web components can be either servlets or JSP pages. *Servlets* are Java programming language classes that dynamically process requests and construct responses. *JSP pages* are text-based documents that execute as servlets but allow a more natural approach to creating static content.

Static HTML pages and applets are bundled with Web components during application assembly, but are not considered Web components by the J2EE specification. Server-side utility classes can also be bundled with Web components and, like HTML pages, are not considered Web components.

Like the client tier and as shown in Figure 1–3, the Web tier might include a JavaBeans component to manage the user input and send that input to enterprise beans running in the business tier for processing.

Business Components

Business code, which is logic that solves or meets the needs of a particular business domain such as banking, retail, or finance, is handled by enterprise beans running in the business tier. Figure 1–4 shows how an enterprise bean receives data from client programs, processes it (if necessary), and sends it to the enter-

prise information system tier for storage. An enterprise bean also retrieves data from storage, processes it (if necessary), and sends it back to the client program.

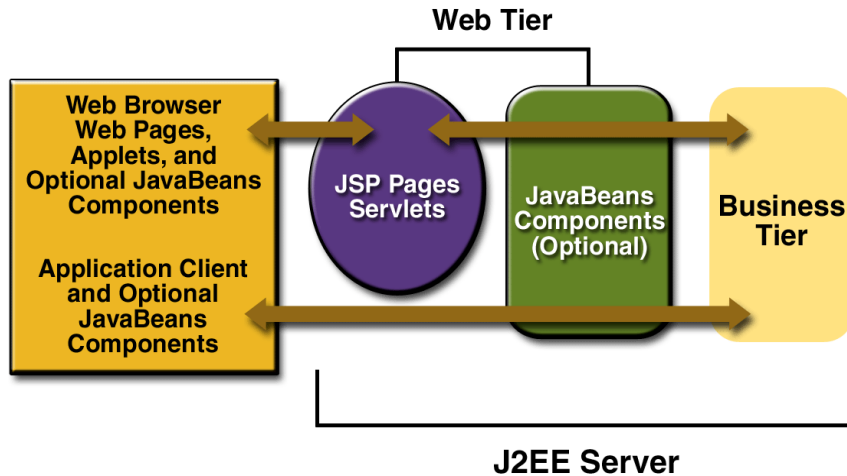


Figure 1–3 Web Tier and J2EE Applications

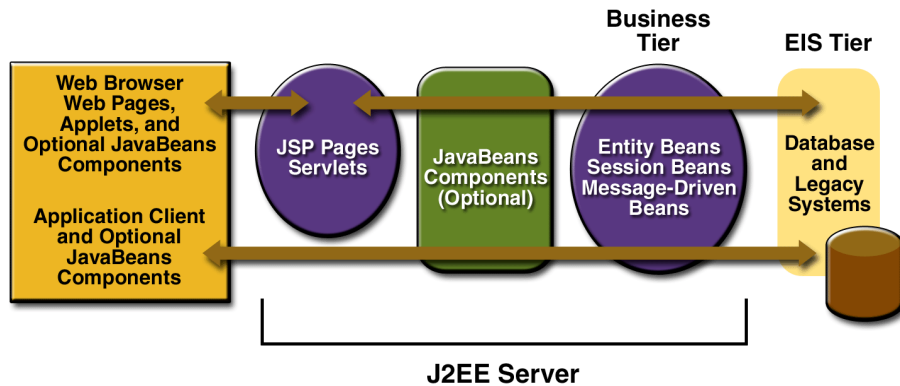


Figure 1–4 Business and EIS Tiers

There are three kinds of enterprise beans: session beans, entity beans, and message-driven beans. A *session bean* represents a transient conversation with a client. When the client finishes executing, the session bean and its data are gone. In contrast, an *entity bean* represents persistent data stored in one row of a database

table. If the client terminates or if the server shuts down, the underlying services ensure that the entity bean data is saved.

A *message-driven bean* combines features of a session bean and a Java Message Service (“JMS”) message listener, allowing a business component to receive JMS messages asynchronously. This tutorial describes entity beans and session beans. For information on message-driven beans, see *The Java Message Service Tutorial*, available at

<http://java.sun.com/products/jms/tutorial/index.html>

Enterprise Information System Tier

The enterprise information system tier handles enterprise information system software and includes enterprise infrastructure systems such as enterprise resource planning (ERP), mainframe transaction processing, database systems, and other legacy information systems. J2EE application components might need access to enterprise information systems for database connectivity, for example.

J2EE Containers

Normally, thin-client multitiered applications are hard to write because they involve many lines of intricate code to handle transaction and state management, multithreading, resource pooling, and other complex low-level details. The component-based and platform-independent J2EE architecture makes J2EE applications easy to write because business logic is organized into reusable components. In addition, the J2EE server provides underlying services in the form of a container for every component type. Because you do not have to develop these services yourself, you are free to concentrate on solving the business problem at hand.

Container Services

Containers are the interface between a component and the low-level platform-specific functionality that supports the component. Before a Web, enterprise bean, or application client component can be executed, it must be assembled into a J2EE application and deployed into its container.

The assembly process involves specifying container settings for each component in the J2EE application and for the J2EE application itself. Container settings customize the underlying support provided by the J2EE server, which includes services such as security, transaction management, Java Naming and Directory Interface™ (JNDI) lookups, and remote connectivity. Here are some of the highlights:

- The J2EE security model lets you configure a Web component or enterprise bean so that system resources are accessed only by authorized users.
- The J2EE transaction model lets you specify relationships among methods that make up a single transaction so that all methods in one transaction are treated as a single unit.
- JNDI lookup services provide a unified interface to multiple naming and directory services in the enterprise so that application components can access naming and directory services.
- The J2EE remote connectivity model manages low-level communications between clients and enterprise beans. After an enterprise bean is created, a client invokes methods on it as if it were in the same virtual machine.

The fact that the J2EE architecture provides configurable services means that application components within the same J2EE application can behave differently based on where they are deployed. For example, an enterprise bean can have security settings that allow it a certain level of access to database data in one production environment and another level of database access in another production environment.

The container also manages nonconfigurable services such as enterprise bean and servlet life cycles, database connection resource pooling, data persistence, and access to the J2EE platform APIs described in the section J2EE APIs (page 17). Although data persistence is a nonconfigurable service, the J2EE architecture lets you override container-managed persistence by including the appropriate code in your enterprise bean implementation when you want more control than the default container-managed persistence provides. For example, you might use bean-managed persistence to implement your own finder (search) methods or to create a customized database cache.

Container Types

The deployment process installs J2EE application components in the J2EE containers illustrated in Figure 1–5.

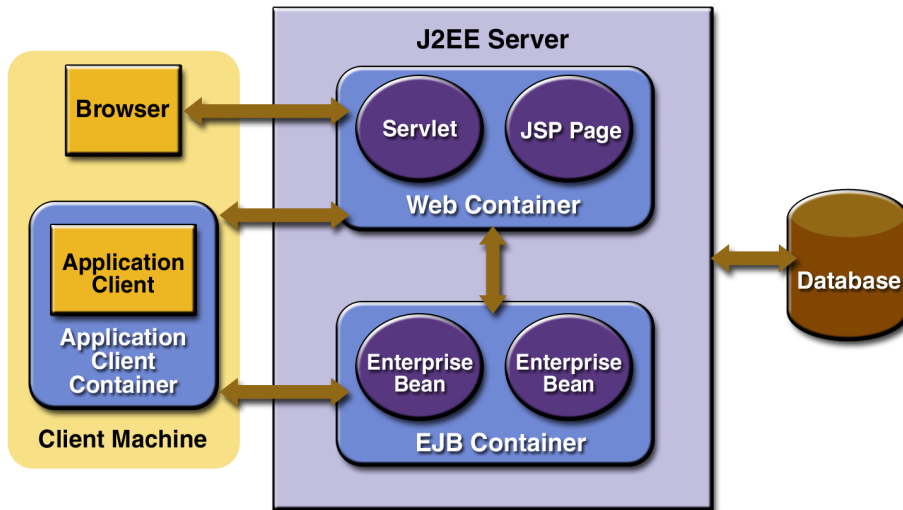


Figure 1-5 J2EE Server and Containers

J2EE server

The runtime portion of a J2EE product. A J2EE server provides EJB and Web containers.

Enterprise JavaBeans (EJB) container

Manages the execution of enterprise beans for J2EE applications. Enterprise beans and their container run on the J2EE server.

Web container

Manages the execution of JSP page and servlet components for J2EE applications. Web components and their container run on the J2EE server.

Application client container

Manages the execution of application client components. Application clients and their container run on the client.

Applet container

Manages the execution of applets. Consists of a Web browser and Java Plug-in running on the client together.

Packaging

A J2EE application is delivered in an Enterprise Archive (EAR) file. An EAR file is a standard Java Archive (JAR) file with an .ear extension. The EAR file

contains J2EE modules. Using EAR files and modules makes it possible to assemble a number of different J2EE applications using some of the same components. No extra coding is needed; it is just a matter of assembling various J2EE modules into J2EE EAR files.

A *J2EE module* consists of one or more J2EE components for the same container type and one component deployment descriptor of that type. A *deployment descriptor* is an XML document with an `.xml` extension that describes a component's deployment settings. An enterprise bean module deployment descriptor, for example, declares transaction attributes and security authorizations for an enterprise bean. Because deployment descriptor information is declarative, it can be changed without modifying the bean source code. At run time, the J2EE server reads the deployment descriptor and acts upon the component accordingly. A J2EE module without an application deployment descriptor can be deployed as a *stand-alone* module. The four types of J2EE modules are:

- Enterprise JavaBeans modules contain class files for enterprise beans and an EJB deployment descriptor. EJB modules are packaged as JAR files with a `.jar` extension.
- Web modules contain JSP files, class files for servlets, GIF and HTML files, and a Web deployment descriptor. Web modules are packaged as JAR files with a `.war` (Web ARchive) extension.
- Resource adapter modules contain all Java interfaces, classes, native libraries, and other documentation, along with the resource adapter deployment descriptor. Together, these implement the Connector architecture (see J2EE Connector Architecture, page 22) for a particular EIS. Resource adapter modules are packaged as JAR files with a `.rar` (Resource adapter ARchive) extension.
- Application client modules contain class files and an application client deployment descriptor. Application client modules are packaged as JAR files with a `.jar` extension.

Development Roles

Reusable modules make it possible to divide the application development and deployment process into distinct roles so that different people or companies can perform different parts of the process.

The first two roles involve purchasing and installing the J2EE product and tools. Once software is purchased and installed, J2EE components can be developed by

application component providers, assembled by application assemblers, and deployed by application deployers. In a large organization, each of these roles might be executed by different individuals or teams. This division of labor works because each of the earlier roles outputs a portable file that is the input for a subsequent role. For example, in the application component development phase, an enterprise bean software developer delivers EJB JAR files. In the application assembly role, another developer combines these EJB JAR files into a J2EE application and saves it in an EAR file. In the application deployment role, a system administrator at the customer site uses the EAR file to install the J2EE application into a J2EE server.

The different roles are not always executed by different people. If you work for a small company, for example, or if you are prototyping a sample application, you might perform the tasks in every phase.

J2EE Product Provider

The J2EE product provider is the company that designs and makes available for purchase the J2EE platform, APIs, and other features defined in the J2EE specification. Product providers are typically operating system, database system, application server, or Web server vendors who implement the J2EE platform according to the Java 2 Platform, Enterprise Edition Specification.

Tool Provider

The tool provider is the company or person who creates development, assembly, and packaging tools used by component providers, assemblers, and deployers.

Application Component Provider

The application component provider is the company or person who creates Web components, enterprise beans, applets, or application clients for use in J2EE applications.

Enterprise Bean Developer

An enterprise bean developer performs the following tasks to deliver an EJB JAR file that contains the enterprise bean:

- Writes and compiles the source code
- Specifies the deployment descriptor
- Bundles the `.class` files and deployment descriptor into an EJB JAR file

Web Component Developer

A Web component developer performs the following tasks to deliver a WAR file containing the Web component:

- Writes and compiles servlet source code
- Writes JSP and HTML files
- Specifies the deployment descriptor for the Web component
- Bundles the `.class`, `.jsp`, `.html`, and deployment descriptor files in the WAR file

J2EE Application Client Developer

An application client developer performs the following tasks to deliver a JAR file containing the J2EE application client:

- Writes and compiles the source code
- Specifies the deployment descriptor for the client
- Bundles the `.class` files and deployment descriptor into the JAR file

Application Assembler

The application assembler is the company or person who receives application component JAR files from component providers and assembles them into a J2EE application EAR file. The assembler or deployer can edit the deployment descriptor directly or use tools that correctly add XML tags according to interactive selections. A software developer performs the following tasks to deliver an EAR file containing the J2EE application:

- Assembles EJB JAR and WAR files created in the previous phases into a J2EE application (EAR) file

- Specifies the deployment descriptor for the J2EE application
- Verifies that the contents of the EAR file are well formed and comply with the J2EE specification

Application Deployer and Administrator

The application deployer and administrator is the company or person who configures and deploys the J2EE application, administers the computing and networking infrastructure where J2EE applications run, and oversees the runtime environment. Duties include such things as setting transaction controls and security attributes and specifying connections to databases.

During configuration, the deployer follows instructions supplied by the application component provider to resolve external dependencies, specify security settings, and assign transaction attributes. During installation, the deployer moves the application components to the server and generates the container-specific classes and interfaces.

A deployer/system administrator performs the following tasks to install and configure a J2EE application:

- Adds the J2EE application (EAR) file created in the preceding phase to the J2EE server
- Configures the J2EE application for the operational environment by modifying the deployment descriptor of the J2EE application
- Verifies that the contents of the EAR file are well formed and comply with the J2EE specification
- Deploys (installs) the J2EE application EAR file into the J2EE server

J2EE APIs

The Sun ONE Application Server provides the following APIs to be used in J2EE applications.

Enterprise JavaBeans Technology

An Enterprise JavaBeans™ (EJB™) component or *enterprise bean* is a body of code with fields and methods to implement modules of business logic. You can

think of an enterprise bean as a building block that can be used alone or with other enterprise beans to execute business logic on the J2EE server.

There are three kinds of enterprise beans: session beans, entity beans, and message-driven beans. Enterprise beans often interact with databases. One of the benefits of entity beans is that you do not have to write any SQL code or use the JDBC™ API directly to perform database access operations; the EJB container handles this for you. However, if you override the default container-managed persistence for any reason, you will need to use the JDBC API. Also, if you choose to have a session bean access the database, you have to use the JDBC API.

JDBC API

The JDBC™ API lets you invoke SQL commands from Java programming language methods. You use the JDBC API in an enterprise bean when you override the default container-managed persistence or have a session bean access the database. With container-managed persistence, database access operations are handled by the container, and your enterprise bean implementation contains no JDBC code or SQL commands. You can also use the JDBC API from a servlet or JSP page to access the database directly without going through an enterprise bean.

The JDBC API has two parts: an application-level interface used by the application components to access a database, and a service provider interface to attach a JDBC driver to the J2EE platform.

Java Servlet Technology

Java Servlet technology lets you define HTTP-specific servlet classes. A servlet class extends the capabilities of servers that host applications accessed by way of a request-response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by Web servers.

JavaServer Pages Technology

JavaServer Pages™ (JSP™) technology lets you put snippets of servlet code directly into a text-based document. A JSP page is a text-based document that

contains two types of text: static template data, which can be expressed in any text-based format such as HTML, WML, and XML, and JSP elements, which determine how the page constructs dynamic content.

Java Message Service

The Java Message Service (JMS) is a messaging standard that allows J2EE application components to create, send, receive, and read messages. It enables distributed communication that is loosely coupled, reliable, and asynchronous. For more information on JMS, see the online Java Message Service Tutorial:

<http://java.sun.com/products/jms/tutorial/index.html>

Java Naming and Directory Interface

The Java Naming and Directory Interface™ (JNDI) provides naming and directory functionality. It provides applications with methods for performing standard directory operations, such as associating attributes with objects and searching for objects using their attributes. Using JNDI, a J2EE application can store and retrieve any type of named Java object.

Because JNDI is independent of any specific implementations, applications can use JNDI to access multiple naming and directory services, including existing naming and directory services such as LDAP, NDS, DNS, and NIS. This allows J2EE applications to coexist with legacy applications and systems. For more information on JNDI, see the online JNDI Tutorial:

<http://java.sun.com/products/jndi/tutorial/index.html>

Java Transaction API

The Java Transaction API (JTA) provides a standard interface for demarcating transactions. The J2EE architecture provides a default auto commit to handle transaction commits and rollbacks. An auto commit means that any other applications viewing data will see the updated data after each database read or write operation. However, if your application performs two separate database access operations that depend on each other, you will want to use the JTA API to demarcate where the entire transaction, including both operations, begins, rolls back, and commits.

JavaMail API

J2EE applications can use the JavaMail™ API to send e-mail notifications. The JavaMail API has two parts: an application-level interface used by the application components to send mail, and a service provider interface. The J2EE platform includes JavaMail with a service provider that allows application components to send Internet mail.

JavaBeans Activation Framework

The JavaBeans Activation Framework (JAF) is included because JavaMail uses it. It provides standard services to determine the type of an arbitrary piece of data, encapsulate access to it, discover the operations available on it, and create the appropriate JavaBeans component to perform those operations.

Java API for XML Processing

The Java API for XML Processing (JAXP) supports the processing of XML documents using Document Object Model (DOM), Simple API for XML Parsing (SAX), and XML Stylesheet Language Transformation (XSLT). JAXP enables applications to parse and transform XML documents independent of a particular XML processing implementation.

JAXP also provides namespace support, which lets you work with schemas that might otherwise have naming conflicts. Designed to be flexible, JAXP lets you use any XML-compliant parser or XSL processor from within your application and supports the W3C schema. You can find information on the W3C schema at this URL: <http://www.w3.org/XML/Schema>.

Java API for XML Registries

The Java API for XML Registries (JAXR) lets you access business and general-purpose registries over the Web. JAXR supports the ebXML Registry/Repository standards and the emerging UDDI specifications. By using JAXR, developers can learn a single API and get access to both of these important registry technologies.

Additionally, businesses submit material to be shared and search for material that others have submitted. Standards groups have developed schemas for partic-

ular kinds of XML documents, and two businesses might, for example, agree to use the schema for their industry's standard purchase order form. Because the schema is stored in a standard business registry, both parties can use JAXR to access it.

Java API for XML-Based RPC

The Java API for XML-based RPC (JAX-RPC) uses the SOAP standard and HTTP so client programs can make XML-based remote procedure calls (RPCs) over the Internet. JAX-RPC also supports WSDL so you can import and export WSDL documents. With JAX-RPC and a WSDL, you can easily interoperate with clients and services running on Java-based or non-Java-based platforms such as .NET. For example, based on the WSDL document, a Visual Basic .NET client can be configured to use a Web service implemented in Java technology or a Web service can be configured to recognize a VB NET client.

JAX-RPC relies on the HTTP transport protocol. Taking that a step further, JAX-RPC lets you create service applications that combine HTTP with a Java technology version of the Secure Socket Layer (SSL) and Transport Layer Security (TLS) protocols to establish basic or mutual authentication. SSL and TLS ensure message integrity by providing data encryption with client and server authentication capabilities.

Authentication is a measured way to verify whether a party is eligible and able to access certain information as a way to protect against the fraudulent use of a system and/or the fraudulent transmission of information. Information transported across the Internet is especially vulnerable to being intercepted and misused, so configuring a JAX-RPC Web service to protect data in transit is very important.

SOAP with Attachments API for Java (SAAJ)

The SOAP with Attachments API for Java (SAAJ) is a low-level API upon which JAX-RPC depends. It enables the production and consumption of messages that conform to the SOAP 1.1 specification and SOAP with Attachments note. Most developers will not use the SAAJ API, but will use the higher-level JAX-RPC API instead.

Note: Java API for XML Messaging (JAXM) is not part of J2EE 1.4. Instead, you can use JAX-RPC and SAAJ to send XML documents over the Web.

J2EE Connector Architecture

The J2EE Connector architecture is used by J2EE tools vendors and system integrators to create resource adapters that support access to enterprise information systems that can be plugged into any J2EE product. A *resource adapter* is a software component that allows J2EE application components to access and interact with the underlying resource manager. Because a resource adapter is specific to its resource manager, there is typically a different resource adapter for each type of database or enterprise information system.

JAX-RPC and the J2EE Connector Architecture are complementary technologies for enterprise application integration (EAI) and end-to-end business integration.

The J2EE Connector Architecture also provides a performance-oriented, secure, scalable, and message-based transactional integration of J2EE-based Web services with existing EISs that can be either synchronous or asynchronous. Existing applications and EISs integrated through the J2EE Connector Architecture into the J2EE platform can be exposed as XML-based Web services using JAX-RPC and J2EE component models.

Java Authentication and Authorization Service

The Java Authentication and Authorization Service (“JAAS”) provides a way for a J2EE application to authenticate and authorize a specific user or group of users to run it.

JAAS is a Java programming language version of the standard Pluggable Authentication Module (PAM) framework that extends the Java 2 Platform security architecture to support user-based authorization.

Simplified Systems Integration

The J2EE platform is a platform-independent, full systems integration solution that creates an open marketplace in which every vendor can sell to every customer. Such a marketplace encourages vendors to compete, not by trying to lock customers into their technologies but by trying to outdo each other by providing products and services that benefit customers, such as better performance, better tools, or better customer support.

The J2EE APIs enable systems and applications integration through the following:

- Unified application model across tiers with enterprise beans
- Simplified response and request mechanism with JSP pages and servlets
- Reliable security model with JAAS
- XML-based data interchange integration with JAXP
- Simplified interoperability with the J2EE Connector Architecture
- Easy database connectivity with the JDBC API
- Enterprise application integration with message-driven beans and JMS, JTA, and JNDI

You can learn more about using the J2EE platform to build integrated business systems by reading *J2EE Technology in Practice*:

<http://java.sun.com/j2ee/inpractice/aboutthebook.html>

Introduction to Web Applications

Stephanie Bodoff

A Web application is a dynamic extension of a Web server. There are two types of Web applications:

- Presentation-oriented. A presentation-oriented Web application generates dynamic Web pages containing various types of markup language (HTML, XML, and so on) in response to requests.
- Service-oriented. A service-oriented Web application implements the end-point of a fine-grained Web service. Service-oriented Web applications are often invoked by presentation-oriented applications.

In the Java 2 Platform, *Web components* provide the dynamic extension capabilities for a Web server. Web components are either Java Servlets or JSP pages. Servlets are Java programming language classes that dynamically process requests and construct responses. JSP pages are text-based documents that execute as servlets but allow a more natural approach to creating static content. Although servlets and JSP pages can be used interchangeably, each has its own strengths. Servlets are best suited to service-oriented Web applications and managing the control functions of a presentation-oriented application, such as dispatching requests and handling nontextual data. JSP pages are more appropriate for generating text-based markup such as HTML, SVG, WML, and XML.

Web components are supported by the services of a runtime platform called a *Web container*. The Web container provides services such as request dispatching,

security, concurrency, and life cycle management. It also gives Web components access to APIs such as naming, transactions, and e-mail.

Certain aspects of Web application behavior can be configured when the application is deployed to the Web container. The configuration information is maintained in a text file in XML format called a *Web application deployment descriptor*. A deployment descriptor must conform to the schema described in the Java Servlet specification.

This chapter describes the organization, configuration, and installation and deployment procedures for Web applications. Chapters 9 and 10 cover how to develop Web components for service-oriented Web applications. Chapters 3 and 4 cover how to develop the Web components for presentation-oriented Web applications. Many features of JSP technology are determined by Java Servlet technology, so you should familiarize yourself with that material even if you do not intend to write servlets.

Most Web applications use the HTTP protocol, and support for HTTP is a major aspect of Web components. For a brief summary of HTTP protocol features see HTTP Overview (page 427).

Web Application Life Cycle

A Web application consists of Web components, static resource files such as images, and helper classes and libraries. The Web container provides many supporting services that enhance the capabilities of Web components and make them easier to develop. However, because it must take these services into account, the process for creating and running a Web application is different from that of traditional stand-alone Java classes.

The process for creating, deploying, and executing a Web application can be summarized as follows:

1. Develop the Web component code (including possibly a deployment descriptor).
2. Build the Web application components along with any static resources (for example, images) and helper classes referenced by the component.
3. Install or deploy the application into a Web container.
4. Access a URL that references the Web application.

Developing Web component code is covered in the later chapters. Steps 2 through 4 are expanded on in the following sections and illustrated with a Hello,

World style presentation-oriented application. This application allows a user to enter a name into an HTML form (Figure 2–1) and then displays a greeting after the name is submitted (Figure 2–2):

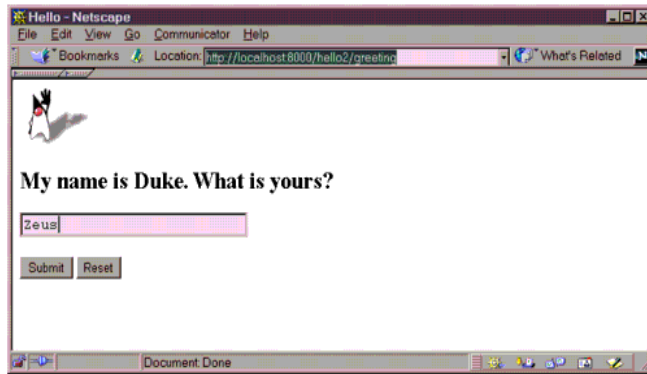


Figure 2–1 Greeting Form

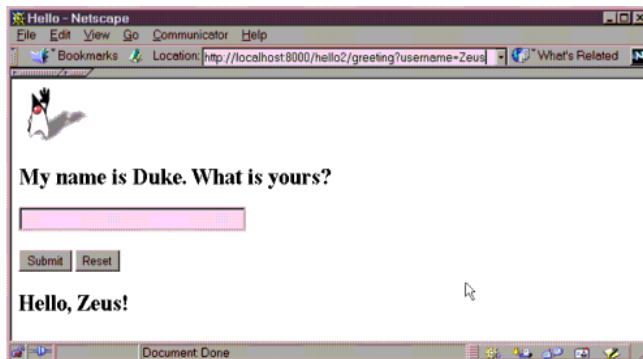


Figure 2–2 Response

The Hello application contains two Web components that generate the greeting and the response. This tutorial has two versions of the application: a servlet version called Hello1, in which the components are implemented by two servlet classes, `GreetingServlet.java` and `ResponseServlet.java`, and a JSP version called Hello2, in which the components are implemented by two JSP pages, `greeting.jsp` and `response.jsp`. The two versions are used to illustrate the tasks involved in packaging, deploying, and running an application that contains Web components. If you are viewing this tutorial online, you must download the

tutorial bundle to get the source code for this example. See About the Examples (page ix).

Web Modules

Web components and static Web content files such as images are called *Web resources*. A *Web module* is the smallest deployable and usable unit of Web resources in a J2EE application. A J2EE Web module corresponds to a *Web application* as defined in the Java Servlet Specification.

Web modules are typically packaged and deployed as Web archive (WAR) files. The format of a WAR file is identical to that of a JAR file. However, the contents and use of WAR files differ from JAR files, so WAR file names use a `.war` extension.

In addition to Web components and Web resources, a Web module can contain other files including:

- Server-side utility classes (database beans, shopping carts, and so on). Often these classes conform to the JavaBeans component architecture.
- Client-side classes (applets and utility classes)

The top-level directory of a Web module is the *document root* of the application. The document root is where JSP pages, *client-side* classes and archives, and static Web resources are stored.

The document root contains a subdirectory called `WEB-INF`, which contains the following files and directories:

- `web.xml` - The Web application deployment descriptor
- Tag library descriptor files (see Tag Library Descriptors, page 122)
- `classes` - A directory that contains *server-side classes*: servlets, utility classes, and JavaBeans components
- `lib` - A directory that contains JAR archives of libraries called by server-side classes

You can also create application-specific subdirectories (that is, package directories) in either the document root or the `WEB-INF/classes` directory.

Creating a Web Module

When you develop Web applications using the IDE, it creates the necessary Web module structure for you. It also creates or modifies the Web application deployment descriptor based on information entered into wizards and property sheets.

To create a Web module in the IDE:

1. Select File→New. This opens the new wizard.
2. Scroll to the JSP & Servlet node and expand it.
3. Select Web module.
4. Click Next.
5. Type the path to the directory for the Web module.
6. Click Finish.

The Web module will be mounted as a filesystem in the IDE. Expand the filesystem and the WEB-INF node to view the Web module structure. If you click on the `web.xml` file, you can view the deployment properties associated with the Web module.

To view the Hello1 Web module:

1. In the IDE, mount the filesystem
`<INSTALL>/j2eetutorial/examples/web/hello1`.
2. Expand the nodes `hello1`→`WEB-INF`.

Creating a Web Component

You create a Web component in the context of a Web module. To create a Web component in the IDE:

1. Mount the Web module as a filesystem.
2. Expand the module and WEB-INF nodes.
3. To create a JSP page, right-click the module node and choose New→JSP & Servlet→JSP or Servlet.
4. To create a servlet, right-click the `classes` node and choose New→JSP & Servlet→Servlet.
5. Type a name for the JSP page or servlet.
6. Click Finish.

Configuring Web Modules

Web applications are configured via elements contained in the Web application deployment descriptor. The IDE generates the descriptor when you create a Web module and adds elements when you create Web components and associated classes. You can modify the elements via the property sheets associated with the descriptor.

The following sections give a brief introduction to the Web application features you will usually want to configure. A number of security parameters can be specified; these are covered in a future release of the tutorial. For a complete listing and description of the features, see the Java Servlet specification.

In the following sections, some examples demonstrate procedures for configuring the Hello, World application. If Hello, World does not use a specific configuration feature, the section gives uses other examples for illustrating the deployment descriptor element and describes generic procedures for specifying the feature using the IDE. Extended examples that demonstrate how to use the IDE are in The Example Servlets (page 48) and The Example JSP Pages (page 150).

Request Mapping

When a request is received by the Sun application server it must determine which Web component should handle the request. It does so by mapping the URL path contained in the request to a Web application and a Web component. A URL path contains the context path and a servlet path

`http://<host>:80/context_path/servlet_path`

A *context path* identifies a Web application. For example, to view the context of the Hello1 application:

1. Select the WEB-INF node of the Hello1 Web module.
2. Note that the Context Root property value is set to /hello1.

The *servlet path* identifies the Web component that should handle a request. The servlet path must start with a / and end with a string or a wildcard expression with an extension (*.jsp, for example). Since Web containers automatically map a servlet path that ends with *.jsp, you do not have to specify a servlet path for a JSP page unless you wish to refer to the page by a name other than its file

name. In the Hello2 example, the greeting page has a servlet path, but `response.jsp` is called by name.

To view the servlet path for the Hello2 application:

1. In the IDE, mount the filesystem
`<INSTALL>/j2eetutorial/examples/web/hello2`.
2. Expand the nodes `hello2`.
3. Select the `greeting` JSP page.
4. Select the Deployment Entries property and open the property editor.
5. Notice that the JSP page `/greeting.jsp` has the servlet name `greeting` and is mapped to the URL `/greeting`.

Initialization Parameters

The Web components in a Web module share an object that represents their application context (see *Accessing the Web Context*, page 75). You can pass initialization parameters to the context or Web component.

To add a context parameter in the IDE:

1. Mount the Web module as a filesystem and expand it.
2. Expand the `WEB-INF` node.
3. Select the `web.xml` file.
4. Select the Deployment tab in the property editor.
5. Select the Context Parameters property and open the property editor.
6. Click Add.
7. Type the parameter name and value.
8. Click OK twice.

For an example context parameter, page mapping, see *The Example JSP Pages* (page 150).

To add a Web component initialization parameter in the IDE:

1. Mount the Web module as a filesystem and expand it.
2. Select the Web component.
3. Select the Deployment Entries property and open the property editor.
4. Click Edit.
5. Under the Init Parameters table, click Add.

6. Type the parameter name and value.
7. Click OK twice.

Error Mappings

You can specify a mapping between the status code returned in an HTTP response or a Java programming language exception returned by any Web component and a Web resource (see Handling Errors, page 54). To set up the mapping, you must specify the Error Pages property for the Web deployment descriptor. To add an error mapping in the IDE:

1. Mount the Web module as a filesystem.
2. Expand the WEB-INF node.
3. Select the web.xml file.
4. Select the Deployment tab in the property editor.
5. Select the Error Pages property and open the property editor.
6. Click Add.
7. Add an HTTP Error Code (see HTTP Responses, page 428) or Java Exception Type.
8. Enter the name of a resource to be invoked when the status code or exception is returned in the Error Page Field. The name should have a leading forward slash /.

Note: You can also define error pages for a JSP page contained in a Web module. If error pages are defined for both the Web module and a JSP page, the JSP page's error page takes precedence.

For an example error page mapping, see The Example Servlets (page 48).

References to Environment Entries, Resource Environment Entries, or Resources

If your Web components reference environment entries, resource environment entries, or resources such as databases, you must declare the references with in the Web application deployment descriptor. To add a reference in the IDE:

1. Mount the Web module as a filesystem.
2. Expand the WEB-INF node.

3. Select the `web.xml` file.
4. In the properties sheet, select the References tab.
5. Click the Resource References Property and open the property editor.
6. Click Add.
7. Type a JNDI name for the resource.
8. Choose the type of the resource.
9. Choose whether the container or the application performs authentication when the resource is accessed.
10. Choose whether the resource can be shared by more than one Web application.

For an example resource reference, see *Configuring the Web Application to Reference a Data Source with JNDI* (page 44).

Deploying Web Modules

Before a Web application can be accessed, it must be deployed as a Web module in the application server. For example, to deploy the `hello1` Web module in the IDE:

1. Start the Sun ONE Application Server 7.
2. Verify that the Sun ONE Application Server 7 has been installed.
 - a. In the Runtime pane of the Explorer, choose the node Server Registry→Installed Servers.
 - b. Verify that the Sun ONE Application Server 7 is listed below Installed Servers. If it is not listed:
 1. Install the Sun ONE Application Server Plugin as described in *Sun™ ONE Studio 4, Enterprise Edition for Java™ with Application Server 7 Tutorial*.
 2. Right-click Sun ONE Application Server 7 in the Installed Servers list and choose Add Admin Server.
 3. Type the host and port (4848) of the application server's admin server.
 4. Type the user name and password you provided when you installed the application server and click OK.
 5. Right-click the admin server just created and choose Create a Server Instance.
 6. Type the host and port (80) of the application server and click OK.

3. Verify that the Sun ONE Application Server 7 is the default server.
 - a. In the Runtime pane of the Explorer, expand the nodes Server Registry→Default Servers.
 - b. Verify that beneath Default Servers there is a node for Web Tier Applications: `server1(host:port)`. If this node is not displayed:
 - 1.Right-click Web Tier Applications and choose Set Default Server.
 - 2.Select the server instance you created in step 2. and click OK.
4. In the IDE, mount the filesystem
`<INSTALL>/j2eetutorial/examples/web/hello1`.
5. Expand the `hello1` node.
6. Right-click the `WEB-INF` directory and choose Deploy.

Listing Deployed Web Modules

You can list deployed Web modules with the application server administration tool or the IDE.

To list all Web modules currently deployed on the application server with the administration tool:

1. Open the URL `http://localhost:4848/admin` in a browser.
2. Select the `server1` node.
3. Expand the nodes Applications→Web Apps.

To list the Web applications deployed on a server with the IDE:

1. Select the Runtime tab in the Explorer.
2. Expand the nodes Server Registry→Installed Servers→Sun ONE Application Server 7→`localhost:4848`→`server1`→Deployed Web Modules.

Running Web Applications

A Web application is executed when a Web browser references a URL that is mapped to component. Once you have installed or deployed the `Hello1` application, you can run the Web application by pointing a browser at

`http://<host>:80/hello1/greeting`

Replace `<host>` with the name of the host running the application server. If your browser is running on the same host as the application server, you may replace `<host>` with `localhost`.

Updating Web Modules

During development, you will often need to make changes to Web applications. The process for viewing those changes is to:

1. Recompile the servlet class.
2. Redeploy the application in the server.
3. Reload the URL in the client.

To try this feature, modify the servlet version of the Hello application. For example, you could change the greeting returned by `GreetingServlet` to be:

```
<h2>Hi, my name is Duke. What's yours?</h2>
```

To update the file:

1. Edit the file `<INSTALL>/j2eetutorial/examples/web/hello1/WEB-INF/classes/GreetingServlet.java`.
2. Redeploy the Web module.
3. Reload the URL in the browser.

You should see the screen in Figure 2–3 in the browser:

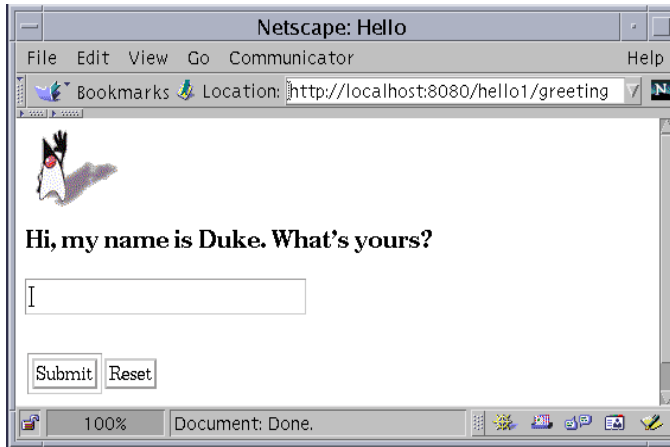


Figure 2–3 New Greeting

Undeploying Web Modules

You can undeploy a Web module with the application server administration tool or the IDE.

To undeploy a Web module with the application server administration tool:

1. Open the URL `http://localhost:4848/admin` in a browser.
2. Select the `server1` node.
3. Expand the `Applications` node.
4. Select the `Web Apps` node.
5. Select the checkbox next to `hello1` and click the `Undeploy` button.

To undeploy a Web application with the IDE:

1. Select the `Runtime` tab in the `Explorer`.
2. Expand the nodes `Server Registry`→`Installed Servers`→`Sun ONE Application Server 7`→`localhost:4848`→`server1`→`Deployed Web Modules`.
3. Right-click `hello1` and choose `Undeploy`.

Duke's Bookstore

In the next 5 chapters, a common example—Duke's Bookstore—is used to illustrate the elements of Java Servlet technology, JavaServer Pages technology, and the JSP Standard Tag Library. The example emulates a simple online shopping application. It provides a book catalog from which users can select books and add them to a shopping cart. Users can view and modify the shopping cart. Once users are finished shopping, they can purchase the books in the cart.

The Duke's Bookstore examples share common classes and a database schema. These files are located in the directory `<INSTALL>/j2eetutorial/examples/web/bookstore`. The common classes are packaged into a JAR and already included in the `WEB-INF/lib` directory in each version of Duke's Bookstore example. To recreate the bookstore library JAR:

1. In the IDE, mount the filesystem
`<INSTALL>/j2eetutorial/examples/web/bookstore`.
2. Expand the bookstore node.
3. Right-click the bookstore node and choose Compile.
4. Create the bookstore JAR.
 - a. Right-click the bookstore node and choose New→JAR Packager→JAR Recipe.
 - b. Type bookstore for the Recipe Name.
 - c. Click Next.
 - d. Select the `cart`, `database`, `exception`, and `messages` packages and click Add.
 - e. Click Next twice.
 - f. Click Generate to generate the JAR manifest.
 - g. Click Finish.
5. Right-click the bookstore JAR recipe node and choose Compile.
6. Expand the bookstore JAR recipe node.

Internationalizing and Localizing Web Applications

Internationalization is the process of preparing an application to support various languages and data formats. *Localization* is the process of adapting an internationalized application to support a specific language or locale. Although all client user interfaces should be internationalized and localized, it is particularly important for Web applications because of the far-reaching nature of the Web. For a good overview of internationalization and localization, see

<http://java.sun.com/docs/books/tutorial/i18n/index.html>

There are two approaches to internationalizing a Web application:

- Provide a version of the JSP page in each of the target locales and have a controller servlet dispatch the request to the appropriate page (depending on the requested locale). This approach is useful if large amounts of data on a page or an entire Web application need to be internationalized.
- Isolate any locale-sensitive data on a page (such as error messages, string literals, or button labels) into resource bundles, and access the data so that the corresponding translated message is fetched automatically and inserted into the page. Thus, instead of creating strings directly in your code, you create a resource bundle that contains translations and read the translations from that bundle using the corresponding key. A resource bundle can be backed by a text file (properties resource bundle) or a class (list resource bundle) containing the mappings.

In the following chapters on Web technology, the Duke's Bookstore example is internationalized and localized into English and Spanish. The key and value pairs are contained in list resource bundles named `messages.BookMessage_*.class`. To give you an idea of what the key and string pairs in a resource bundle look like, here are a few lines from the file `messages.BookMessages.java`.

```
{"TitleCashier", "Cashier"},
{"TitleBookDescription", "Book Description"},
{"Visitor", "You are visitor number "},
{"What", "What We're Reading"},
{"Talk", " talks about how Web components can transform the way
you develop applications for the Web. This is a must read for
any self respecting Web developer!"},
{"Start", "Start Shopping"},

```

To get the correct strings for a given user, a Web component retrieves the locale (set by a browser language preference) from the request, opens the resource bundle for that locale, and then saves the bundle as a session attribute (see Associating Attributes with a Session, page 76):

```
ResourceBundle messages = (ResourceBundle)session.  
    getAttribute("messages");  
if (messages == null) {  
    Locale locale=request.getLocale();  
    messages = ResourceBundle.getBundle("WebMessages",  
        locale);  
    session.setAttribute("messages", messages);  
}
```

A Web component retrieves the resource bundle from the session:

```
ResourceBundle messages =  
    (ResourceBundle)session.getAttribute("messages");
```

and looks up the string associated with the key `TitleCashier` as follows:

```
messages.getString("TitleCashier");
```

This has been a very brief introduction to internationalizing Web applications. For more information on this subject see the Java BluePrints:

<http://java.sun.com/blueprints>

Accessing Databases from Web Applications

Data that is shared between Web components and persistent between invocations of a Web application is usually maintained in a database. Web applications use the JDBC 2.0 API to access relational databases. In the JDBC API, databases are accessed via `DataSource` objects. A `DataSource` has a set of properties that identify and describe the real world data source that it represents. These properties include information like the location of the database server, the name of the database, the network protocol to use to communicate with the server, and so on.

Applications access a data source using a connection, and a `DataSource` object can be thought of as a factory for connections to the particular data source that the `DataSource` instance represents. In a basic `DataSource` implementation, a

call to the method `DataSource.getConnection` returns a connection object that is a physical connection to the data source.

If a `DataSource` object is registered with a JNDI naming service, an application can use the JNDI API to access that `DataSource` object, which can then be used to connect to the data source it represents.

`DataSource` objects that implement connection pooling also produce a connection to the particular data source that the `DataSource` class represents. The connection object that the method `DataSource.getConnection` returns is a handle to a `PooledConnection` object rather than being a physical connection. An application uses the connection object just as it usually does and is generally unaware that it is in any way different. Connection pooling has no effect whatever on application code except that a pooled connection, as is true with all connections, should always be explicitly closed. When an application closes a connection that is pooled, the connection is returned to a pool of reusable connections. The next time `DataSource.getConnection` is called, a handle to one of these pooled connections will be returned if one is available. Because connection pooling avoids creating a new physical connection every time one is requested, it can help to make applications run significantly faster.

The Duke's Bookstore examples use the PointBase database shipped with Sun ONE Application Server 7 to maintain the catalog of books. This section describes how to:

- Start the PointBase database server
- Populate the database
- Add the PointBase JDBC driver to the application server's classpath
- Define a data source in the application server
- Configure a Web application to reference the data source with a JNDI name
- Map the JNDI name to the data source defined in the application server

Starting the PointBase Database Server

The Sun ONE Application Server 7 is distributed with a development version of the PointBase database server. To start the server:

1. Open a terminal window.
2. Navigate to `<S1AS7_HOME>/pointbase/server`.
3. Execute `StartServer`.

Populating the Example Database

The Sun ONE Application Server 7 is distributed with a sample database that contains the table required for the Duke's Bookstore examples. The table is accessible only to the user name BOOKSTORE. If you need to repopulate the database:

1. Start the PointBase console tool:
 - a. Open a terminal window.
 - b. Navigate to `<S1AS7_HOME>/pointbase/client_tools`.
 - c. Execute `PB_console`.
2. In the PointBase console window, connect to the database `jdbc:pointbase:server://localhost/sun-appserv-samples` with user name `PBPUBLIC` and password `PBPUBLIC`.
3. In the IDE, mount the filesystem `<INSTALL>/j2eetutorial/examples/web/bookstore`.
4. Expand the bookstore node.
5. In the PointBase console window, execute the SQL statements in `createUser.sql`:
 - a. Copy the SQL statements in `createUser.sql`.
 - b. Paste the SQL statements into the text area labeled Enter SQL Commands by choosing Edit→Paste in the PointBase console window.
 - c. Choose SQL→Execute All.
6. Choose DBA→Disconnect from Database.
7. Choose DBA→Connect to Database.
8. Type `BOOKSTORE` for the user name and `BOOKSTORE` for the password.
9. In the PointBase console window, execute the SQL statements in `bookstore.sql`. At the end of the processing, you should see the following output:

```
[java] ID
[java] -----
[java] 201
[java] 202
[java] 203
[java] 204
[java] 205
[java] 206
[java] 207
```

```
[java]
[java] 7 Rows Selected.
[java]
[java] SQL>
[java]
[java] COMMIT;
[java] OK
```

Add PointBase JDBC Driver to the Application Server's Classpath

You can add the PointBase JDBC driver to the application server's classpath in one of two ways:

- Copy the PointBase JDBC driver library `<S1AS7_HOME>/pointbase/client_tools/lib/pbclient42RE.jar` to the `lib/` directory of your application server instance. For example: `<S1AS7_HOME>/domains/domain1/server1/lib/`.
- Specify the location of the PointBase driver in the Classpath Suffix field in the application server's configuration:
 - a. Start the administration console by opening the URL `http://localhost:4848` in a browser.
 - b. Select the `server1` node.
 - c. Select the JVM Settings tab.
 - d. Click the Path Settings link.
 - e. Add `<S1AS7_HOME>/pointbase/client_tools/lib/pbclient42RE.jar` to the Classpath Suffix text area.
 - f. Click Save.

Then, restart the application server to make the server aware of the driver:

1. Click the General tab of the administration console.
2. Click Stop to stop the server and Start to restart it.

Defining a Data Source in Sun ONE Application Server 7

Data sources in the Sun ONE Application Server 7 implement connection pooling. Thus, to define the Duke's Bookstore data source, you first need to define a data pool as follows:

1. In the IDE, select the Runtime tab of the Explorer.
2. Expand the nodes Server Registry→Installed Servers→Sun ONE Application Server 7→localhost:4848.
3. Right-click Unregistered JDBC Connection Pools and select Add New JDBC Data Pool.
4. Type `com.pointbase.jdbc.jdbcDataSource` for the DataSource Class-name.
5. Type `bookstore-pool` for the Name.
6. Click Properties and open the property editor.
7. Add the properties and values listed in Table 2-1:

Table 2-1 Bookstore Pool Properties

Property	Value
DatabaseName	<code>jdbc:pointbase:server://localhost/sun-appserv-samples</code>
User	BOOKSTORE
Password	BOOKSTORE

8. Right-click the `bookstore-pool` data pool and choose Register.
9. Select the application server that you want the pool to be registered in and click Register.
10. Click OK.
11. Expand the Registered JDBC Connection Pools node and notice that `bookstore-pool` is listed.

Then, create the data source as follows:

1. In the IDE, select the Runtime tab of the Explorer.

2. Expand the nodes Server Registry→Installed Servers→Sun ONE Application Server 7→localhost:4848.
3. Right-click Unregistered JDBC Data Sources and select Add a new JDBC Data Source.
4. Type jdbc/BookDB for the JNDI Name.
5. Choose bookstore-pool for the Pool Name.
6. Right-click the jdbc/BookDB data source and choose Register.
7. Select the application server that you want the data source to be registered in and click Register.
8. Click OK.
9. Expand the Registered JDBC Connection Data Sources node and notice that jdbc/BookDB is listed.

Configuring the Web Application to Reference a Data Source with JNDI

In order to access a database from a Web application, you must declare resource reference in the application's Web application deployment descriptor (see References to Environment Entries, Resource Environment Entries, or Resources, page 32). The resource reference declares a JNDI name, the type of the data resource, and the kind of authentication used when the resource is accessed. The JNDI name is used to create a data source object in the database helper class `database.BookDB`:

```
public BookDB () throws Exception {
    try {
        Context initCtx = new InitialContext();
        Context envCtx = (Context)
            initCtx.lookup("java:comp/env");
        DataSource ds = (DataSource) envCtx.lookup("jdbc/BookDB");
        con = ds.getConnection();
        System.out.println("Created connection to database.");
    } catch (Exception ex) {
        System.out.println("Couldn't create connection." +
            ex.getMessage());
        throw new
            Exception("Couldn't open connection to database: "
                +ex.getMessage());
    }
}
```


To specify a resource reference to the bookstore data source:

1. Select the Web module.
2. Expand the WEB-INF node.
3. Select the web.xml.
4. Select the References tab in the property sheet.
5. Click the Resource References Property and open the property editor.
6. Click Add.
7. Type jdbc/BookDB in the Name field.
8. Leave the default type javax.sql.DataSource.
9. Leave the default authorization Container.
10. Choose Shareable for Sharing Scope.

Mapping the Web Application JNDI Name to a Data Source

Since the resource reference declared in the Web application deployment descriptor uses a JNDI name to refer to the data source, you must connect the name to a data source defined by the application server as follows:

1. Select the Web module.
2. Expand the WEB-INF node.
3. Select the web.xml file.
4. Select the Sun ONE AS property sheet.
5. Select the Resource Reference Mapping property and open the property editor.
6. For the Resource Reference Name jdbc/BookDB, type jdbc/BookDB in the JNDI Name field.

Further Information

For more information about Web applications, refer to the following:

- Resources listed on the Web site <http://java.sun.com/products/servlet>.
- The Java Servlet 2.3 Specification.

- *Getting Started with Sun™ ONE Application Server 7* located at `<S1AS7_HOME>/docs/getting-started/ch6-database-setup.html` contains further information on how to set up the application server to access a database.

Java Servlet Technology

Stephanie Bodoff

AS soon as the Web began to be used for delivering services, service providers recognized the need for dynamic content. Applets, one of the earliest attempts toward this goal, focused on using the client platform to deliver dynamic user experiences. At the same time, developers also investigated using the server platform for this purpose. Initially, Common Gateway Interface (CGI) scripts were the main technology used to generate dynamic content. Though widely used, CGI scripting technology has a number of shortcomings, including platform dependence and lack of scalability. To address these limitations, Java Servlet technology was created as a portable way to provide dynamic, user-oriented content.

What is a Servlet?

A *servlet* is a Java programming language class used to extend the capabilities of servers that host applications accessed via a request-response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by Web servers. For such applications, Java Servlet technology defines HTTP-specific servlet classes.

The `javax.servlet` and `javax.servlet.http` packages provide interfaces and classes for writing servlets. All servlets must implement the `Servlet` interface, which defines life-cycle methods.

When implementing a generic service, you can use or extend the `GenericServlet` class provided with the Java Servlet API. The `HttpServlet` class provides methods, such as `doGet` and `doPost`, for handling HTTP-specific services.

This chapter focuses on writing servlets that generate responses to HTTP requests. Some knowledge of the HTTP protocol is assumed; if you are unfamiliar with this protocol, you can get a brief introduction to HTTP in HTTP Overview (page 427).

The Example Servlets

This chapter uses the Duke's Bookstore application to illustrate the tasks involved in programming servlets. Table 3-1 lists the servlets that handle each bookstore function. Each programming task is illustrated by one or more servlets. For example, `BookDetailsServlet` illustrates how to handle HTTP GET requests, `BookDetailsServlet` and `CatalogServlet` show how to construct responses, and `CatalogServlet` illustrates how to track session information.

Table 3-1 Duke's Bookstore Example Servlets

Function	Servlet
Enter the bookstore	<code>BookStoreServlet</code>
Create the bookstore banner	<code>BannerServlet</code>
Browse the bookstore catalog	<code>CatalogServlet</code>
Put a book in a shopping cart	<code>CatalogServlet</code> , <code>BookDetailsServlet</code>
Get detailed information on a specific book	<code>BookDetailsServlet</code>
Display the shopping cart	<code>ShowCartServlet</code>
Remove one or more books from the shopping cart	<code>ShowCartServlet</code>
Buy the books in the shopping cart	<code>CashierServlet</code>

Table 3–1 Duke’s Bookstore Example Servlets (Continued)

Function	Servlet
Receive an acknowledgement for the purchase	ReceiptServlet

The data for the bookstore application is maintained in a database and accessed through the helper class `database.BookDB`. The database package also contains the class `BookDetails`, which represents a book. The shopping cart and shopping cart items are represented by the classes `cart.ShoppingCart` and `cart.ShoppingCartItem`, respectively.

The source for the Duke’s Bookstore application is located in the `<INSTALL>/j2eetutorial/examples/web/bookstore1` directory created when you unzip the tutorial bundle (see About the Examples, page ix).

To deploy and run the example:

1. In the IDE, mount the filesystem
`<INSTALL>/j2eetutorial/examples/web/bookstore1`.
2. Expand the `bookstore1` node.
3. Right-click the `WEB-INF` directory and choose `Deploy`.
4. Set up the PointBase database as described in *Accessing Databases from Web Applications*, page 39.
5. Open the bookstore URL `http://localhost:80/bookstore1/enter`.

To review the deployment settings:

1. Expand the `WEB-INF` node.
2. Select the `web.xml` file.
3. Select the `Deployment` property sheet.
4. Browse the error page mappings.
 - a. Click the `Error Pages` property and open the property editor.
 - b. Notice that two HTTP status codes (404 and 500) and four exceptions are mapped to the error page `/errorpage.html`.
5. Browse the filter property and mappings.
 - a. Click the `Filters` property and open the property editor.

- b. Notice that the filters classes are `filters.HitCounterFilter` and `filters.OrderFilter` and they are mapped to `BookStoreServlet` and `ReceiptServlet` respectively.
6. Browse the listener property.
 - a. Click the Listeners property and open the property editor.
 - b. Notice that the listener class is `listeners.ContextListener`.
7. Browse the servlet definitions and their mappings.
 - a. Click the Servlets property and open the property editor.
 - b. Notice that there are 7 servlets listed defined and mapped to URLs.
8. Browse the resource references.
 - a. Select the Resources Property sheet.
 - a. Click the Resource References property and open the property editor.
 - b. Note the resource reference named `jdbc/BookDB`.

Troubleshooting

Because we have specified an error page, you will see the message `The application is unavailable. Please try later, when an exception occurs.` You will have to look in the application server's log to determine the cause of the exception. The log file is `<S1AS7_HOME>/domains/domain1/server1/logs/server.log`. You can view the log in the application server administration tool as follows:

1. Start the administration console by opening the URL `http://localhost:4848` in a browser.
2. Select the `server1` node.
3. Select the Logging tab.
4. Click the View Event Log link.

Here are the exceptions that can be returned and their causes:

- `UnavailableException`—Returned if a servlet can't retrieve the Web context attribute representing the bookstore database. This will occur if the PointBase server hasn't been started. In the log you will see the following:
`12/Feb/2003:12:28:14] INFO (4944): CORE3282: stdout: Couldn't create connection.SQL-server rejected establishment of SQL-connection. PointBase.`
`[12/Feb/2003:12:28:14] INFO (4944): CORE3282: stdout:`

Couldn't create databaseCouldn't open connection to database: SQL-server rejected establishment of SQL-connection. PointBase.

If you have the database server running but the application server can't communicate with it due to firewall interference, then you will see the following error:

```
[12/Feb/2003:16:15:14] INFO ( 1952): CORE3282: stdout:
Couldn't create connection.null
[12/Feb/2003:16:15:14] INFO ( 1952): CORE3282: stdout:
Couldn't create databaseCouldn't open connection to database: null
```

If you have not defined a data source that references the PointBase database (see *Defining a Data Source in Sun ONE Application Server 7*, page 43), you will see the following:

```
[13/Feb/2003:10:32:23] INFO ( 3128): CORE3282: stdout:
Couldn't create connection.BookDB not found
```

- **BookNotFoundException**—Returned if a book can't be located in the bookstore database. This will occur if the PointBase database server has crashed.
- **BooksNotFoundException**—Returned if the bookstore data can't be retrieved. This will occur if the PointBase database server has crashed.

Servlet Life Cycle

The life cycle of a servlet is controlled by the container in which the servlet has been deployed. When a request is mapped to a servlet, the container performs the following steps.

1. If an instance of the servlet does not exist, the Web container
 - a. Loads the servlet class.
 - b. Creates an instance of the servlet class.
 - c. Initializes the servlet instance by calling the `init` method. Initialization is covered in *Initializing a Servlet* (page 58).
2. Invokes the `service` method, passing a request and response object. Service methods are discussed in *Writing Service Methods* (page 59).

If the container needs to remove the servlet, it finalizes the servlet by calling the servlet's `destroy` method. Finalization is discussed in *Finalizing a Servlet* (page 79).

Handling Servlet Life Cycle Events

You can monitor and react to events in a servlet's life cycle by defining listener objects whose methods get invoked when life cycle events occur. To use listener objects you

- Define the listener class
- Specify the listener class in the Web application deployment descriptor

Defining The Listener Class

You define a listener class as an implementation of a listener interface. Servlet Life Cycle Events (page 52) lists the events that can be monitored and the corresponding interface that must be implemented. When a listener method is invoked, it is passed an event that contains information appropriate to the event. For example, the methods in the `HttpSessionListener` interface are passed an `HttpSessionEvent`, which contains an `HttpSession`.

Table 3–2 Servlet Life Cycle Events

Object	Event	Listener Interface and Event Class
Web context (See Accessing the Web Context, page 75)	Initialization and destruction	<code>javax.servlet.</code> <code>ServletContextListener</code> and <code>ServletContextEvent</code>
	Attribute added, removed, or replaced	<code>javax.servlet.</code> <code>ServletContextAttributeListener</code> and <code>ServletContextAttributeEvent</code>
Session (See Maintaining Client State, page 76)	Creation, invalidation, and timeout	<code>javax.servlet.http.</code> <code>HttpSessionListener</code> and <code>HttpSessionEvent</code>
	Attribute added, removed, or replaced	<code>javax.servlet.http.</code> <code>HttpSessionAttributeListener</code> and <code>HttpSessionBindingEvent</code>

The `listeners.ContextListener` class creates and removes the database helper and counter objects used in the Duke's Bookstore application. The meth-

ods retrieve the Web context object from `ServletContextEvent` and then store (and remove) the objects as servlet context attributes.

```
import database.BookDB;
import javax.servlet.*;
import util.Counter;

public final class ContextListener
    implements ServletContextListener {
    private ServletContext context = null;
    public void contextInitialized(ServletContextEvent event) {
        context = event.getServletContext();
        try {
            BookDB bookDB = new BookDB();
            context.setAttribute("bookDB", bookDB);
        } catch (Exception ex) {
            System.out.println(
                "Couldn't create database: "
                + ex.getMessage());
        }
        Counter counter = new Counter();
        context.setAttribute("hitCounter", counter);
        context.log("Created hitCounter"
            + counter.getCounter());
        counter = new Counter();
        context.setAttribute("orderCounter", counter);
        context.log("Created orderCounter"
            + counter.getCounter());
    }

    public void contextDestroyed(ServletContextEvent event) {
        context = event.getServletContext();
        BookDB bookDB = context.getAttribute(
            "bookDB");
        bookDB.remove();
        context.removeAttribute("bookDB");
        context.removeAttribute("hitCounter");
        context.removeAttribute("orderCounter");
    }
}
```

Creating a Listener

To create a listener in the IDE:

1. Mount the Web module as a filesystem.

2. Expand the module and WEB-INF nodes.
3. Right-click the `classes` node and choose New→JSP & Servlet→Listeners→XXXListener, where XXX is the type of the listener.
4. Type a name for the listener.
5. Click Finish.

Specifying Event Listener Classes

To specify an event listener in Web application deployment descriptor using the IDE:

1. Mount the Web module as a filesystem.
2. Expand the WEB-INF node.
3. Select the `web.xml` file.
4. Select the Deployment tab in the property editor.
5. Select the Listeners property and open the property editor.
6. Click Add.
7. Type the listener class.
8. Click OK twice.

For an example listener definition, see The Example Servlets (page 48).

Handling Errors

Any number of exceptions can occur when a servlet is executed. The Web container will generate a default page containing the message `A Servlet Exception Has Occurred` when an exception occurs, but you can also specify that the container should return a specific error page for a given exception. To specify such a page, you specify an Error Pages property for the Web application deployment descriptor (see Error Mappings, page 32).

Sharing Information

Web components, like most objects, usually work with other objects to accomplish their tasks. There are several ways they can do this. They can use private helper objects (for example, JavaBeans components), they can share objects that are attributes of a public scope, they can use a database, and they can invoke

other Web resources. The Java Servlet technology mechanisms that allow a Web component to invoke other Web resources are described in *Invoking Other Web Resources* (page 72).

Using Scope Objects

Collaborating Web components share information via objects maintained as attributes of four scope objects. These attributes are accessed with the `[get|set]Attribute` methods of the class representing the scope. Table 3–3 lists the scope objects.

Table 3–3 Scope Objects

Scope Object	Class	Accessible From
Web context	<code>javax.servlet.ServletContext</code>	Web components within a Web context. See <i>Accessing the Web Context</i> (page 75).
session	<code>javax.servlet.http.HttpSession</code>	Web components handling a request that belongs to the session. See <i>Maintaining Client State</i> (page 76).
request	subtype of <code>javax.servlet.ServletException</code>	Web components handling the request.
page	<code>javax.servlet.jsp.PageContext</code>	The JSP page that creates the object. See <i>Implicit Objects</i> (page 92).

Figure 3–1 shows the scoped attributes maintained by the Duke’s Bookstore application.

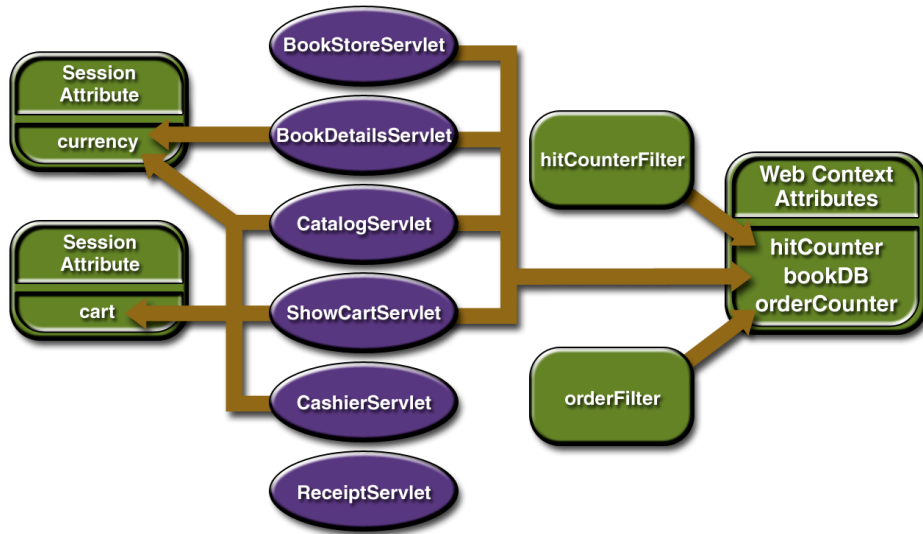


Figure 3–1 Duke’s Bookstore Scoped Attributes

Controlling Concurrent Access to Shared Resources

In a multithreaded server, it is possible for shared resources to be accessed concurrently. Besides scope object attributes, shared resources include in-memory data such as instance or class variables, and external objects such as files, database connections, and network connections. Concurrent access can arise in several situations:

- Multiple Web components accessing objects stored in the Web context
- Multiple Web components accessing objects stored in a session
- Multiple threads within a Web component accessing instance variables. A Web container will typically create a thread to handle each request. If you want to ensure that a servlet instance handles only one request at a time, a servlet can implement the `SingleThreadModel` interface. If a servlet implements this interface, you are guaranteed that no two threads will execute concurrently in the servlet’s service method. A Web container can

implement this guarantee by synchronizing access to a single instance of the servlet, or by maintaining a pool of Web component instances and dispatching each new request to a free instance.

Note: This interface does not prevent synchronization problems that result from Web components accessing shared resources such as static class variables or external objects.

When resources can be accessed concurrently, they can be used in an inconsistent fashion. To prevent this, you must control the access using the synchronization techniques described in the Threads lesson in *The Java Tutorial*.

In the previous section we showed five scoped attributes shared by more than one servlet: `bookDB`, `cart`, `currency`, `hitCounter`, and `orderCounter`. The `bookDB` attribute is discussed in the next section. The `cart`, `currency`, and counters can be set and read by multiple multithreaded servlets. To prevent these objects from being used inconsistently, access is controlled by synchronized methods. For example, here is the `util.Counter` class:

```
public class Counter {
    private int counter;
    public Counter() {
        counter = 0;
    }
    public synchronized int getCounter() {
        return counter;
    }
    public synchronized int setCounter(int c) {
        counter = c;
        return counter;
    }
    public synchronized int incCounter() {
        return(++counter);
    }
}
```

Accessing Databases

Data that is shared between Web components and is persistent between invocations of a Web application is usually maintained by a database. Web components use the JDBC 2.0 API to access relational databases. The data for the bookstore application is maintained in a database and accessed through the helper class

database.BookDB. For example, `ReceiptServlet` invokes the `BookDB.buyBooks` method to update the book inventory when a user makes a purchase. The `buyBooks` method invokes `buyBook` for each book contained in the shopping cart. To ensure the order is processed in its entirety, the calls to `buyBook` are wrapped in a single JDBC transaction. The use of the shared database connection is synchronized via the `[get|release]Connection` methods.

```
public void buyBooks(ShoppingCart cart) throws OrderException {
    Collection items = cart.getItems();
    Iterator i = items.iterator();
    try {
        getConnection();
        con.setAutoCommit(false);
        while (i.hasNext()) {
            ShoppingCartItem sci = (ShoppingCartItem)i.next();
            BookDetails bd = (BookDetails)sci.getItem();
            String id = bd.getBookId();
            int quantity = sci.getQuantity();
            buyBook(id, quantity);
        }
        con.commit();
        con.setAutoCommit(true);
        releaseConnection();
    } catch (Exception ex) {
        try {
            con.rollback();
            releaseConnection();
            throw new OrderException("Transaction failed: " +
                ex.getMessage());
        } catch (SQLException sqx) {
            releaseConnection();
            throw new OrderException("Rollback failed: " +
                sqx.getMessage());
        }
    }
}
```

Initializing a Servlet

After the Web container loads and instantiates the servlet class and before it delivers requests from clients, the Web container initializes the servlet. You can customize this process to allow the servlet to read persistent configuration data, initialize resources, and perform any other one-time activities by overriding the

`init` method of the `Servlet` interface. A servlet that cannot complete its initialization process should throw `UnavailableException`.

All the servlets that access the bookstore database (`BookStoreServlet`, `CatalogServlet`, `BookDetailsServlet`, and `ShowCartServlet`) initialize a variable in their `init` method that points to the database helper object created by the Web context listener:

```
public class CatalogServlet extends HttpServlet {
    private BookDB bookDB;
    public void init() throws ServletException {
        bookDB = (BookDB)getContext().
            getAttribute("bookDB");
        if (bookDB == null) throw new
            UnavailableException("Couldn't get database.");
    }
}
```

Writing Service Methods

The service provided by a servlet is implemented in the `service` method of a `GenericServlet`, the `doMethod` methods (where *Method* can take the value `Get`, `Delete`, `Options`, `Post`, `Put`, `Trace`) of an `HttpServlet`, or any other protocol-specific methods defined by a class that implements the `Servlet` interface. In the rest of this chapter, the term *service method* will be used for any method in a servlet class that provides a service to a client.

The general pattern for a service method is to extract information from the request, access external resources, and then populate the response based on that information.

For HTTP servlets, the correct procedure for populating the response is to first fill in the response headers, then retrieve an output stream from the response, and finally write any body content to the output stream. Response headers must always be set before a `PrintWriter` or `ServletOutputStream` is retrieved because the HTTP protocol expects to receive all headers before body content. The next two sections describe how to get information from requests and generate responses.

Getting Information from Requests

A request contains data passed between a client and the servlet. All requests implement the `ServletRequest` interface. This interface defines methods for accessing the following information:

- Parameters, which are typically used to convey information between clients and servlets
- Object-valued attributes, which are typically used to pass information between the servlet container and a servlet or between collaborating servlets
- Information about the protocol used to communicate the request and the client and server involved in the request
- Information relevant to localization

For example, in `CatalogServlet` the identifier of the book that a customer wishes to purchase is included as a parameter to the request. The following code fragment illustrates how to use the `getParameter` method to extract the identifier:

```
String bookId = request.getParameter("Add");
if (bookId != null) {
    BookDetails book = bookDB.getBookDetails(bookId);
}
```

You can also retrieve an input stream from the request and manually parse the data. To read character data, use the `BufferedReader` object returned by the request's `getReader` method. To read binary data, use the `ServletInputStream` returned by `getInputStream`.

HTTP servlets are passed an HTTP request object, `HttpServletRequest`, which contains the request URL, HTTP headers, query string, and so on.

An HTTP request URL contains the following parts:

```
http://[host]:[port][request path]?[query string]
```

The request path is further composed of the following elements:

- **Context path:** A concatenation of a forward slash / with the context root of the servlet's Web application.
- **Servlet path:** The path section that corresponds to the component alias that activated this request. This path starts with a forward slash /.

- **Path info:** The part of the request path that is not part of the context path or the servlet path.

If the context path is `/catalog` and for the aliases listed in Table 3–4, Table 3–5 gives some examples of how the URL will be broken down.

Table 3–4 Aliases

Pattern	Servlet
<code>/lawn/*</code>	<code>LawnServlet</code>
<code>/*.jsp</code>	<code>JSPServlet</code>

Table 3–5 Request Path Elements

Request Path	Servlet Path	Path Info
<code>/catalog/lawn/index.html</code>	<code>/lawn</code>	<code>/index.html</code>
<code>/catalog/help/feedback.jsp</code>	<code>/help/feedback.jsp</code>	<code>null</code>

Query strings are composed of a set of parameters and values. Individual parameters are retrieved from a request with the `getParameter` method. There are two ways to generate query strings:

- A query string can explicitly appear in a Web page. For example, an HTML page generated by the `CatalogServlet` could contain the link `Add To Cart`. `CatalogServlet` extracts the parameter named `Add` as follows:

```
String bookId = request.getParameter("Add");
```

- A query string is appended to a URL when a form with a GET HTTP method is submitted. In the Duke's Bookstore application, `CashierServlet` generates a form, then a user name input to the form is appended to the URL that maps to `ReceiptServlet`, and finally `ReceiptServlet` extracts the user name using the `getParameter` method.

Constructing Responses

A response contains data passed between a server and the client. All responses implement the `ServletResponse` interface. This interface defines methods that allow you to do the following:

- Retrieve an output stream to use to send data to the client. To send character data, use the `PrintWriter` returned by the response's `getWriter` method. To send binary data in a MIME body response, use the `ServletOutputStream` returned by `getOutputStream`. To mix binary and text data, for example, to create a multipart response, use a `ServletOutputStream` and manage the character sections manually.
- Indicate the content type (for example, `text/html`), being returned by the response. A registry of content type names is kept by the Internet Assigned Numbers Authority (IANA) at:

`ftp://ftp.isi.edu/in-notes/iana/assignments/media-types`

- Indicate whether to buffer output. By default, any content written to the output stream is immediately sent to the client. Buffering allows content to be written before anything is actually sent back to the client, thus providing the servlet with more time to set appropriate status codes and headers or forward to another Web resource.
- Set localization information.

HTTP response objects, `HttpServletResponse`, have fields representing HTTP headers such as

- Status codes, which are used to indicate the reason a request is not satisfied.
- Cookies, which are used to store application-specific information at the client. Sometimes cookies are used to maintain an identifier for tracking a user's session (see Session Tracking (page 78)).

In Duke's Bookstore, `BookDetailsServlet` generates an HTML page that displays information about a book that the servlet retrieves from a database. The servlet first sets response headers: the content type of the response and the buffer size. The servlet buffers the page content because the database access can generate an exception that would cause forwarding to an error page. By buffering the response, the client will not see a concatenation of part of a Duke's Bookstore page with the error page should an error occur. The `doGet` method then retrieves a `PrintWriter` from the response.

For filling in the response, the servlet first dispatches the request to `BannerServlet`, which generates a common banner for all the servlets in the application. This process is discussed in Including Other Resources in the Response (page 72). Then the servlet retrieves the book identifier from a request parameter and uses the identifier to retrieve information about the book from the bookstore database. Finally, the servlet generates HTML markup that describes the book information and commits the response to the client by calling the `close` method on the `PrintWriter`.

```
public class BookDetailsServlet extends HttpServlet {
    public void doGet (HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        // set headers before accessing the Writer
        response.setContentType("text/html");
        response.setBufferSize(8192);
        PrintWriter out = response.getWriter();

        // then write the response
        out.println("<html>" +
            "<head><title>+
            messages.getString("TitleBookDescription")
            +</title></head>");

        // Get the dispatcher; it gets the banner to the user
        RequestDispatcher dispatcher =
            getServletContext().
            getRequestDispatcher("/banner");
        if (dispatcher != null)
            dispatcher.include(request, response);

        //Get the identifier of the book to display
        String bookId = request.getParameter("bookId");
        if (bookId != null) {
            // and the information about the book
            try {
                BookDetails bd =
                    bookDB.getBookDetails(bookId);
                ...
                //Print out the information obtained
                out.println("<h2>" + bd.getTitle() + "</h2>" +
                    ...
            } catch (BookNotFoundException ex) {
                response.resetBuffer();
                throw new ServletException(ex);
            }
        }
    }
}
```

```
        out.println("</body></html>");  
        out.close();  
    }  
}
```

BookDetailsServlet generates a page that looks like:



Figure 3–2 Book Details

Filtering Requests and Responses

A *filter* is an object that can transform the header and content (or both) of a request or response. Filters differ from Web components in that they usually do not themselves create a response. Instead, a filter provides functionality that can be “attached” to any kind of Web resource. As a consequence, a filter should not have any dependencies on a Web resource for which it is acting as a filter, so that

it can be composable with more than one type of Web resource. The main tasks that a filter can perform are as follows:

- Query the request and act accordingly.
- Block the request and response pair from passing any further.
- Modify the request headers and data. You do this by providing a customized version of the request.
- Modify the response headers and data. You do this by providing a customized version of the response.
- Interact with external resources.

Applications of filters include authentication, logging, image conversion, data compression, encryption, tokenizing streams, and XML transformations, and so on.

You can configure a Web resource to be filtered by a chain of zero, one, or more filters in a specific order. This chain is specified when the Web application containing the component is deployed and is instantiated when a Web container loads the component.

In summary, the tasks involved in using filters include

- Programming the filter
- Programming customized requests and responses
- Specifying the filter chain for each Web resource

Programming Filters

The filtering API is defined by the `Filter`, `FilterChain`, and `FilterConfig` interfaces in the `javax.servlet` package. You define a filter by implementing the `Filter` interface. The most important method in this interface is the `doFilter` method, which is passed request, response, and filter chain objects. This method can perform the following actions:

- Examine the request headers.
- Customize the request object if it wishes to modify request headers or data.
- Customize the response object if it wishes to modify response headers or data.
- Invoke the next entity in the filter chain. If the current filter is the last filter in the chain that ends with the target Web component or static resource, the next entity is the resource at the end of the chain; otherwise, it is the next

filter that was configured in the WAR. It invokes the next entity by calling the `doFilter` method on the chain object (passing in the request and response it was called with, or the wrapped versions it may have created). Alternatively, it can choose to block the request by not making the call to invoke the next entity. In the latter case, the filter is responsible for filling out the response.

- Examine response headers after it has invoked the next filter in the chain
- Throw an exception to indicate an error in processing

In addition to `doFilter`, you must implement the `init` and `destroy` methods. The `init` method is called by the container when the filter is instantiated. If you wish to pass initialization parameters to the filter, you retrieve them from the `FilterConfig` object passed to `init`.

The Duke's Bookstore application uses the filters `HitCounterFilter` and `OrderFilter` to increment and log the value of a counter when the entry and receipt servlets are accessed.

In the `doFilter` method, both filters retrieve the servlet context from the filter configuration object so that they can access the counters stored as context attributes. After the filters have completed application-specific processing, they invoke `doFilter` on the filter chain object passed into the original `doFilter` method. The elided code is discussed in the next section.

```
public final class HitCounterFilter implements Filter {
    private FilterConfig filterConfig = null;

    public void init(FilterConfig filterConfig)
        throws ServletException {
        this.filterConfig = filterConfig;
    }
    public void destroy() {
        this.filterConfig = null;
    }
    public void doFilter(ServletRequest request,
        ServletResponse response, FilterChain chain)
        throws IOException, ServletException {
        if (filterConfig == null)
            return;
        StringWriter sw = new StringWriter();
        PrintWriter writer = new PrintWriter(sw);
        Counter counter = (Counter)filterConfig.
            getServletContext().
            getAttribute("hitCounter");
        writer.println();
```

```

        writer.println("=====");
        writer.println("The number of hits is: " +
            counter.incCounter());
        writer.println("=====");
        // Log the resulting string
        writer.flush();
        filterConfig.getServletContext().
            log(sw.getBuffer().toString());
        ...
        chain.doFilter(request, wrapper);
        ...
    }
}

```

Programming Customized Requests and Responses

There are many ways for a filter to modify a request or response. For example, a filter could add an attribute to the request or insert data in the response. In the Duke's Bookstore example, `HitCounterFilter` inserts the value of the counter into the response.

A filter that modifies a response must usually capture the response before it is returned to the client. The way to do this is to pass a stand-in stream to the servlet that generates the response. The stand-in stream prevents the servlet from closing the original response stream when it completes and allows the filter to modify the servlet's response.

To pass this stand-in stream to the servlet, the filter creates a response wrapper that overrides the `getWriter` or `getOutputStream` method to return this stand-in stream. The wrapper is passed to the `doFilter` method of the filter chain. Wrapper methods default to calling through to the wrapped request or response object. This approach follows the well-known Wrapper or Decorator pattern described in *Design Patterns, Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1995). The following sections describe how the hit counter filter described earlier and other types of filters use wrappers.

To override request methods, you wrap the request in an object that extends `ServletRequestWrapper` or `HttpServletRequestWrapper`. To override response methods, you wrap the response in an object that extends `ServletResponseWrapper` or `HttpServletResponseWrapper`.

HitCounterFilter wraps the response in a CharResponseWrapper. The wrapped response is passed to the next object in the filter chain, which is BookStoreServlet. BookStoreServlet writes its response into the stream created by CharResponseWrapper. When chain.doFilter returns, HitCounterFilter retrieves the servlet's response from PrintWriter and writes it to a buffer. The filter inserts the value of the counter into the buffer, resets the content length header of the response, and finally writes the contents of the buffer to the response stream.

```
PrintWriter out = response.getWriter();
CharResponseWrapper wrapper = new CharResponseWrapper(
    (HttpServletResponse)response);
chain.doFilter(request, wrapper);
CharArrayWriter caw = new CharArrayWriter();
caw.write(wrapper.toString().substring(0,
    wrapper.toString().indexOf("</body>")-1));
caw.write("<p>\n<center>" +
    messages.getString("Visitor") + "<font color='red'>" +
    counter.getCounter() + "</font></center>");
caw.write("\n</body></html>");
response.setContentLength(caw.toString().length());
out.write(caw.toString());
out.close();

public class CharResponseWrapper extends
    HttpServletResponseWrapper {
    private CharArrayWriter output;
    public String toString() {
        return output.toString();
    }
    public CharResponseWrapper(HttpServletResponse response){
        super(response);
        output = new CharArrayWriter();
    }
    public PrintWriter getWriter(){
        return new PrintWriter(output);
    }
}
```


Figure 3–3 shows the entry page for Duke’s Bookstore with the hit counter.

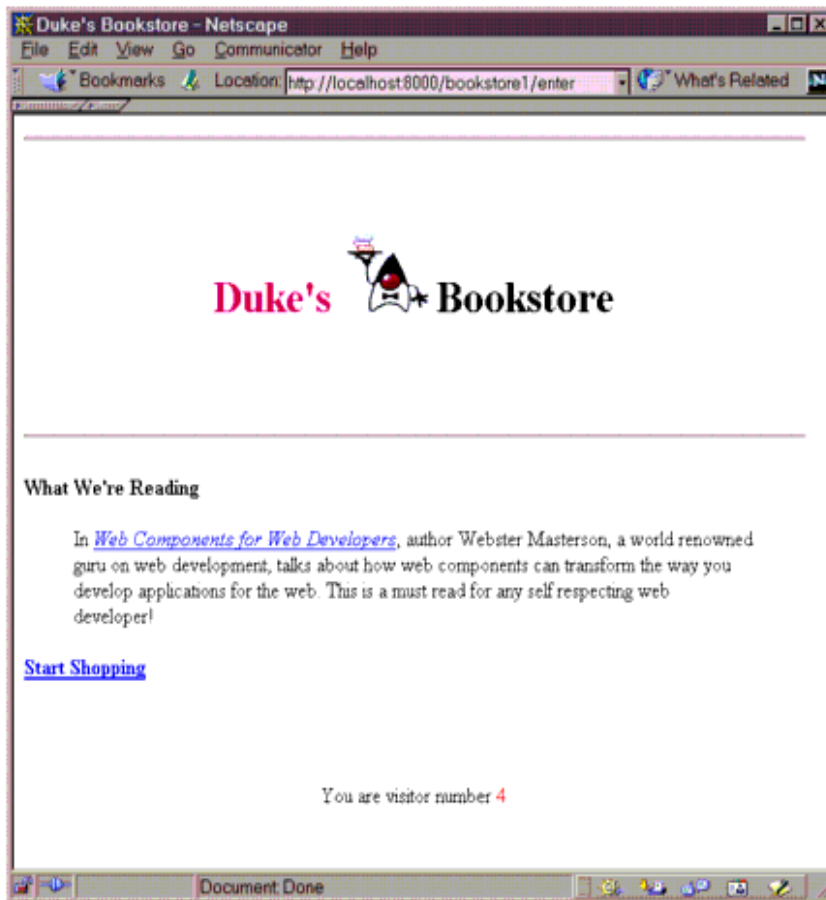


Figure 3–3 Duke’s Bookstore

Creating a Filter

To create a filter in the IDE:

1. Mount the Web module as a filesystem.
2. Expand the module and WEB-INF nodes.
3. Right-click the `classes` node and choose New→JSP & Servlet→Filters→XXXFilter, where XXX is Advanced or Simple. You choose Advanced

if your filter has a customized request or response; otherwise, choose Simple.

4. Type a name for the filter.
5. Click Finish.

Specifying Filter Mappings

A Web container uses filter mappings to decide how to apply filters to Web resources. A filter mapping matches a filter to a Web component by name or to Web resources by URL pattern. The filters are invoked in the order in which filter mappings appear in the filter mapping list of a WAR. You specify a filter mapping list for a Web module by

- Declaring the filter using the Filters property. This property creates a name for the filter and declares the filter's implementation class and initialization parameters.
- Mapping the filter to a Web resource in a Filter Mappings property. This property maps a filter name to a Web resource by name or by URL pattern.

To add a filter and filter mapping properties in the IDE:

1. Mount the Web module as a filesystem.
2. Expand the WEB-INF node.
3. Select the web.xml file.
4. Select the Deployment tab in the property editor.
5. Select the Filters property and open the property editor.
6. Click Add.
7. Type the filter name and filter class.
8. Select the Filter Mappings property and to open the property editor.
9. Select the filter.
10. Click Add.
11. Type a URL pattern or servlet name to which the filter should be applied.
12. Click OK twice.

For an example filter definition and mapping, see The Example Servlets (page 48).

If you want to log every request to a Web application, you would map the hit counter filter to the URL pattern `/*`. Table 3–6 summarizes the filter mapping list

for the Duke's Bookstore application. The filters are matched by URL pattern and each filter chain contains only one filter.

Table 3–6 Duke's Bookstore Filter Mapping List

URL	Filter
/enter	HitCounterFilter
/receipt	OrderFilter

You can map a filter to one or more Web resources and you can map more than one filter to a Web resource. This is illustrated in Figure 3–4, where filter F1 is mapped to servlets S1, S2, and S3, filter F2 is mapped to servlet S2, and filter F3 is mapped to servlets S1 and S2.

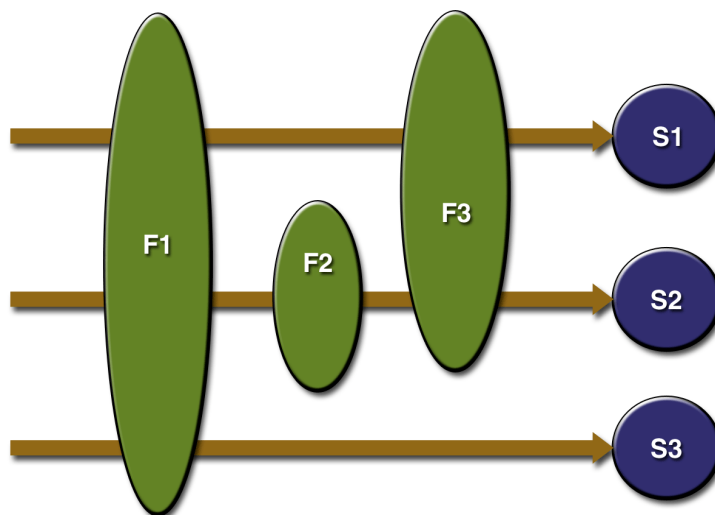


Figure 3–4 Filter to Servlet Mapping

Recall that a filter chain is one of the objects passed to the `doFilter` method of a filter. This chain is formed indirectly via filter mappings. The order of the filters in the chain is the same as the order in which filter mappings appear in the Web application deployment descriptor.

When a filter is mapped to servlet S1, the Web container invokes the `doFilter` method of F1. The `doFilter` method of each filter in S1's filter chain is invoked by the preceding filter in the chain via the `chain.doFilter` method. Since S1's filter chain contains filters F1 and F3, F1's call to `chain.doFilter` invokes the `doFilter` method of filter F3. When F3's `doFilter` method completes, control returns to F1's `doFilter` method.

Invoking Other Web Resources

Web components can invoke other Web resources in two ways: indirect and direct. A Web component indirectly invokes another Web resource when it embeds in content returned to a client a URL that points to another Web component. In the Duke's Bookstore application, most Web components contain embedded URLs that point to other Web components. For example, `ShowCartServlet` indirectly invokes the `CatalogServlet` through the embedded URL `/bookstore1/catalog`.

A Web component can also directly invoke another resource while it is executing. There are two possibilities: it can include the content of another resource, or it can forward a request to another resource.

To invoke a resource available on the server that is running a Web component, you must first obtain a `RequestDispatcher` object using the `getRequestDispatcher("URL")` method.

You can get a `RequestDispatcher` object from either a request or the Web context, however, the two methods have slightly different behavior. The method takes the path to the requested resource as an argument. A request can take a relative path (that is, one that does not begin with a `/`), but the Web context requires an absolute path. If the resource is not available, or if the server has not implemented a `RequestDispatcher` object for that type of resource, `getRequestDispatcher` will return null. Your servlet should be prepared to deal with this condition.

Including Other Resources in the Response

It is often useful to include another Web resource, for example, banner content or copyright information, in the response returned from a Web component. To

include another resource, invoke the `include` method of a `RequestDispatcher` object:

```
include(request, response);
```

If the resource is static, the `include` method enables programmatic server-side includes. If the resource is a Web component, the effect of the method is to send the request to the included Web component, execute the Web component, and then include the result of the execution in the response from the containing servlet. An included Web component has access to the request object, but it is limited in what it can do with the response object:

- It can write to the body of the response and commit a response.
- It cannot set headers or call any method (for example, `setCookie`) that affects the headers of the response.

The banner for the Duke's Bookstore application is generated by `BannerServlet`. Note that both the `doGet` and `doPost` methods are implemented because `BannerServlet` can be dispatched from either method in a calling servlet.

```
public class BannerServlet extends HttpServlet {
    public void doGet (HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {

        PrintWriter out = response.getWriter();
        out.println("<body bgcolor=\"#ffffff\">" +
            "<center>" + "<hr> <br> &nbsp;" + "<h1>" +
            "<font size=\"+3\" color=\"#CC0066\">Duke's </font>" +
            "<img src=\"" + request.getContextPath() +
            "/duke.books.gif\">" +
            "<font size=\"+3\" color=\"black\">Bookstore</font>" +
            "</h1>" + "</center>" + "<br> &nbsp;" + "<hr> <br> ");
    }
    public void doPost (HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {

        PrintWriter out = response.getWriter();
        out.println("<body bgcolor=\"#ffffff\">" +
            "<center>" + "<hr> <br> &nbsp;" + "<h1>" +
            "<font size=\"+3\" color=\"#CC0066\">Duke's </font>" +
            "<img src=\"" + request.getContextPath() +
            "/duke.books.gif\">" +
```

```

        "<font size=\"+3\" color=\"black\">Bookstore</font>" +
        "</h1>" + "</center>" + "<br> &nbsp; <hr> <br> ");
    }
}

```

Each servlet in the Duke's Bookstore application includes the result from `BannerServlet` with the following code:

```

RequestDispatcher dispatcher =
    getServletContext().getRequestDispatcher("/banner");
if (dispatcher != null)
    dispatcher.include(request, response);
}

```

Transferring Control to Another Web Component

In some applications, you might want to have one Web component do preliminary processing of a request and have another component generate the response. For example, you might want to partially process a request and then transfer to another component depending on the nature of the request.

To transfer control to another Web component, you invoke the `forward` method of a `RequestDispatcher`. When a request is forwarded, the request URL is set to the path of the forwarded page. If the original URL is required for any processing, you can save it as a request attribute. The `Dispatcher` servlet, used by a version of the Duke's Bookstore application described in *The Example JSP Pages* (page 114), saves the path information from the original URL, retrieves a `RequestDispatcher` from the request, and then forwards to the JSP page `template.jsp`.

```

public class Dispatcher extends HttpServlet {
    public void doGet(HttpServletRequest request,
        HttpServletResponse response) {
        request.setAttribute("selectedScreen",
            request.getServletPath());
        RequestDispatcher dispatcher = request.
            getRequestDispatcher("/template.jsp");
        if (dispatcher != null)
            dispatcher.forward(request, response);
    }
}

```

```

    }
    public void doPost(HttpServletRequest request,
        ...
    }

```

The forward method should be used to give another resource responsibility for replying to the user. If you have already accessed a `ServletOutputStream` or `PrintWriter` object within the servlet, you cannot use this method; it throws an `IllegalStateException`.

Accessing the Web Context

The context in which Web components execute is an object that implements the `ServletContext` interface. You retrieve the Web context with the `getServletContext` method. The Web context provides methods for accessing:

- Initialization parameters
- Resources associated with the Web context
- Object-valued attributes
- Logging capabilities

The Web context is used by the Duke's Bookstore filters `filters.HitCounterFilter` and `OrderFilter`, which were discussed in *Filtering Requests and Responses* (page 64). The filters store a counter as a context attribute. Recall from *Controlling Concurrent Access to Shared Resources* (page 56) that the counter's access methods are synchronized to prevent incompatible operations by servlets that are running concurrently. A filter retrieves the counter object with the context's `getAttribute` method. The incremented value of the counter is recorded with the context's `log` method.

```

public final class HitCounterFilter implements Filter {
    private FilterConfig filterConfig = null;
    public void doFilter(ServletRequest request,
        ServletResponse response, FilterChain chain)
        throws IOException, ServletException {
        ...
        StringWriter sw = new StringWriter();
        PrintWriter writer = new PrintWriter(sw);
        ServletContext context = filterConfig.
            getServletContext();
        Counter counter = (Counter)context.
            getAttribute("hitCounter");
        ...
    }
}

```

```
        writer.println("The number of hits is: " +
            counter.incCounter());
        ...
        context.log(sw.getBuffer().toString());
        ...
    }
}
```

Maintaining Client State

Many applications require a series of requests from a client to be associated with one another. For example, the Duke's Bookstore application saves the state of a user's shopping cart across requests. Web-based applications are responsible for maintaining such state, called a *session*, because the HTTP protocol is stateless. To support applications that need to maintain state, Java Servlet technology provides an API for managing sessions and allows several mechanisms for implementing sessions.

Accessing a Session

Sessions are represented by an `HttpSession` object. You access a session by calling the `getSession` method of a request object. This method returns the current session associated with this request, or, if the request does not have a session, it creates one. Since `getSession` may modify the response header (if cookies are the session tracking mechanism), it needs to be called before you retrieve a `PrintWriter` or `ServletOutputStream`.

Associating Attributes with a Session

You can associate object-valued attributes with a session by name. Such attributes are accessible by any Web component that belongs to the same Web context *and* is handling a request that is part of the same session.

The Duke's Bookstore application stores a customer's shopping cart as a session attribute. This allows the shopping cart to be saved between requests and also allows cooperating servlets to access the cart. `CatalogServlet` adds items to the cart; `ShowCartServlet` displays, deletes items from, and clears the cart; and `CashierServlet` retrieves the total cost of the books in the cart.


```
public class CashierServlet extends HttpServlet {
    public void doGet (HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {

        // Get the user's session and shopping cart
        HttpSession session = request.getSession();
        ShoppingCart cart =
            (ShoppingCart)session.
                getAttribute("cart");

        ...
        // Determine the total price of the user's books
        double total = cart.getTotal();
    }
}
```

Notifying Objects That Are Associated with a Session

Recall that your application can notify Web context and session listener objects of servlet life cycle events (Handling Servlet Life Cycle Events (page 52)). You can also notify objects of certain events related to their association with a session such as the following:

- When the object is added to or removed from a session. To receive this notification, your object must implement the `javax.http.HttpSessionBindingListener` interface.
- When the session to which the object is attached will be passivated or activated. A session will be passivated or activated when it is moved between virtual machines or saved to and restored from persistent storage. To receive this notification, your object must implement the `javax.http.HttpSessionActivationListener` interface.

Session Management

Since there is no way for an HTTP client to signal that it no longer needs a session, each session has an associated timeout so that its resources can be reclaimed. The timeout period can be accessed with a session's `[get|set]MaxInactiveInterval` methods. You can also set the time-out period in the IDE as follows:

1. Select the `web.xml` file of a Web module.
2. Select the Deployment tab.


```

"<a href=\"\" +
response.encodeURL(request.getContextPath() + "/cashier") +
"\">" + messages.getString("Checkout") +
"</a> &nbsp; &nbsp; &nbsp;" +
"<a href=\"\" +
response.encodeURL(request.getContextPath() +
"/showcart?Clear=clear") +
"\">" + messages.getString("ClearCart") +
"</a></strong>");

```

If cookies are turned off, the session is encoded in the Check Out URL as follows:

```

http://localhost:80/bookstore1/cashier;
jsessionid=c0o7fszeb1

```

If cookies are turned on, the URL is simply

```

http://localhost:80/bookstore1/cashier

```

Finalizing a Servlet

When a servlet container determines that a servlet should be removed from service (for example, when a container wants to reclaim memory resources, or when it is being shut down), it calls the `destroy` method of the `Servlet` interface. In this method, you release any resources the servlet is using and save any persistent state. The following `destroy` method releases the database object created in the `init` method described in *Initializing a Servlet* (page 58):

```

public void destroy() {
    bookDB = null;
}

```

All of a servlet's service methods should be complete when a servlet is removed. The server tries to ensure this by calling the `destroy` method only after all service requests have returned, or after a server-specific grace period, whichever comes first. If your servlet has operations that take a long time to run (that is, operations that may run longer than the server's grace period), the operations could still be running when `destroy` is called. You must make sure that any

threads still handling client requests complete; the remainder of this section describes how to:

- Keep track of how many threads are currently running the service method
- Provide a clean shutdown by having the destroy method notify long-running threads of the shutdown and wait for them to complete
- Have the long-running methods poll periodically to check for shutdown and, if necessary, stop working, clean up, and return

Tracking Service Requests

To track service requests, include in your servlet class a field that counts the number of service methods that are running. The field should have synchronized access methods to increment, decrement, and return its value.

```
public class ShutdownExample extends HttpServlet {
    private int serviceCounter = 0;
    ...
    //Access methods for serviceCounter
    protected synchronized void enteringServiceMethod() {
        serviceCounter++;
    }
    protected synchronized void leavingServiceMethod() {
        serviceCounter--;
    }
    protected synchronized int numServices() {
        return serviceCounter;
    }
}
```

The service method should increment the service counter each time the method is entered and should decrement the counter each time the method returns. This is one of the few times that your `HttpServlet` subclass should override the service method. The new method should call `super.service` to preserve all of the original service method's functionality:

```
protected void service(HttpServletRequest req,
                        HttpServletResponse resp)
                        throws ServletException, IOException {
    enteringServiceMethod();
    try {
        super.service(req, resp);
    }
```

```

    } finally {
        leavingServiceMethod();
    }
}

```

Notifying Methods to Shut Down

To ensure a clean shutdown, your destroy method should not release any shared resources until all of the service requests have completed. One part of doing this is to check the service counter. Another part is to notify the long-running methods that it is time to shut down. For this notification another field is required. The field should have the usual access methods:

```

public class ShutdownExample extends HttpServlet {
    private boolean shuttingDown;
    ...
    //Access methods for shuttingDown
    protected synchronized void setShuttingDown(boolean flag) {
        shuttingDown = flag;
    }
    protected synchronized boolean isShuttingDown() {
        return shuttingDown;
    }
}

```

An example of the destroy method using these fields to provide a clean shutdown follows:

```

public void destroy() {
    /* Check to see whether there are still service methods /*
    /* running, and if there are, tell them to stop. */
    if (numServices() > 0) {
        setShuttingDown(true);
    }

    /* Wait for the service methods to stop. */
    while(numServices() > 0) {
        try {
            Thread.sleep(interval);
        } catch (InterruptedException e) {
        }
    }
}

```

Creating Polite Long-Running Methods

The final step in providing a clean shutdown is to make any long-running methods behave politely. Methods that might run for a long time should check the value of the field that notifies them of shutdowns and should interrupt their work, if necessary.

```
public void doPost(...) {  
    ...  
    for(i = 0; ((i < lotsOfStuffToDo) &&  
        !isShuttingDown()); i++) {  
        try {  
            partOfLongRunningOperation(i);  
        } catch (InterruptedException e) {  
            ...  
        }  
    }  
}
```

Further Information

For further information on Java Servlet technology see:

- Resources listed on the Web site <http://java.sun.com/products/servlet>.
- The Java Servlet 2.3 Specification.

JavaServer Pages Technology

Stephanie Bodoff

JAVASERVER Pages (JSP) technology allows you to easily create Web content that has both static and dynamic components. JSP technology projects all the dynamic capabilities of Java Servlet technology but provides a more natural approach to creating static content. The main features of JSP technology are

- A language for developing JSP pages, which are text-based documents that describe how to process a request and construct a response
- Constructs for accessing server-side objects
- Mechanisms for defining extensions to the JSP language

JSP technology also contains an API that is used by developers of Web containers, but this API is not covered in this chapter.

What Is a JSP Page?

A *JSP page* is a text-based document that contains two types of text: static template data, which can be expressed in any text-based format, such as HTML, SVG, WML, and XML; and JSP elements, which construct dynamic content. A syntax card and reference for the JSP elements are available at

<http://java.sun.com/products/jsp/technical.html#syntax>

The Web page in Figure 4–1 is a form that allows you to select a locale and displays the date in a manner appropriate to the locale.

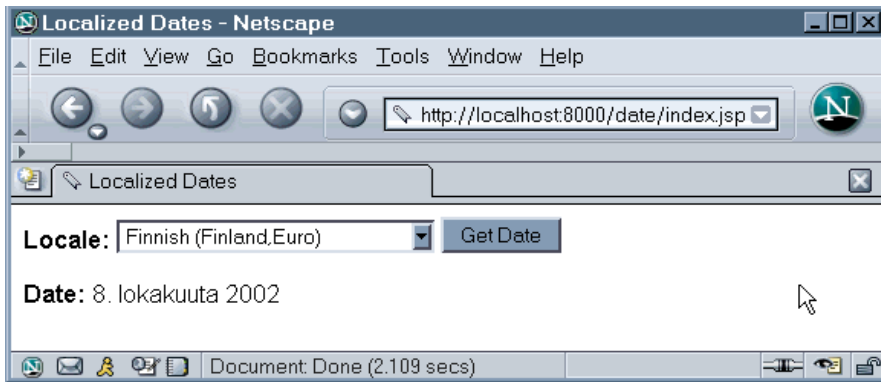


Figure 4–1 Localized Date Form

The source code for this example is in the docs/tutorial/examples/web/date directory created when you unzip the tutorial bundle. The JSP page `index.jsp` used to create the form appears below; it is a typical mixture of static HTML markup and JSP elements. If you have developed Web pages, you are probably familiar with the HTML document structure statements (`<head>`, `<body>`, and so on) and the HTML statements that create a form `<form>` and a menu `<select>`. The lines in bold in the example code contains the following types of JSP constructs:

- Directives (**`<%@page ... %>`**) import classes in the `java.util` package and the `MyLocales` class, and set the content type returned by the page.
- The **`jsp:useBean`** element creates an object containing a collection of locales and initializes a variable that points to that object.
- Scriptlets (**`<% ... %>`**) retrieve the value of the `locale` request parameter, iterate over a collection of locale names, and conditionally insert HTML text into the output.
- Expressions (**`<%= ... %>`**) insert the value of the locale name into the response.

- The **jsp:include** element sends a request to another page (`date.jsp`) and includes the response in the response from the calling page.

```
<%@ page import="java.util.*,MyLocales" %>
<%@ page contentType="text/html; charset=ISO-8859-5" %>
<html>
<head><title>Localized Dates</title></head>
<body bgcolor="white">
<jsp:useBean id="locales" scope="application"
  class="MyLocales"/>
<form name="localeForm" action="index.jsp" method="post">
<b>Locale:</b>
<select name=locale>
<%
  String selectedLocale = request.getParameter("locale");
  Iterator i = locales.getLocaleNames().iterator();
  while (i.hasNext()) {
    String locale = (String)i.next();
    if (selectedLocale != null &&
        selectedLocale.equals(locale)) {
%>
      <option selected><%=locale%></option>
<%
    } else {
%>
      <option><%=locale%></option>
<%
    }
  }
%>
</select>
<input type="submit" name="Submit" value="Get Date">
</form>
<jsp:include page="date.jsp"/>
</body>
</html>
```

The source for the Duke's Bookstore application is located in the `<INSTALL>/j2eetutorial/examples/web/date` directory.

To build, deploy, and execute this JSP page:

1. In the IDE, mount the filesystem `<INSTALL>/j2eetutorial/examples/web/date`.
2. Expand the date node.
3. Right-click the WEB-INF directory and choose Deploy.

4. Right-click the WEB-INF directory and choose Execute.

You will see a combo box whose entries are locales. Select a locale and click Get Date. You will see the date expressed in a manner appropriate for that locale.

The Example JSP Pages

To illustrate JSP technology, this chapter rewrites each servlet in the Duke's Bookstore application introduced in The Example Servlets (page 48) as a JSP page:

Table 4-1 Duke's Bookstore Example JSP Pages

Function	JSP Pages
Enter the bookstore	bookstore.jsp
Create the bookstore banner	banner.jsp
Browse the books offered for sale	catalog.jsp
Put a book in a shopping cart	catalog.jsp and bookdetails.jsp
Get detailed information on a specific book	bookdetails.jsp
Display the shopping cart	showcart.jsp
Remove one or more books from the shopping cart	showcart.jsp
Buy the books in the shopping cart	cashier.jsp
Receive an acknowledgement for the purchase	receipt.jsp

The data for the bookstore application is still maintained in a database. However, two changes are made to the database helper object `database.BookDB`:

- The database helper object is rewritten to conform to JavaBeans component design patterns as described in JavaBeans Component Design Conventions (page 103). This change is made so that JSP pages can access the helper object using JSP language elements specific to JavaBeans components.

- Instead of accessing the bookstore database directly, the helper object goes through a data access object database.`BookDAO`.

The implementation of the database helper object follows. The bean has two instance variables: the current book and a reference to the database enterprise bean.

```
public class BookDB {
    private String bookId = "0";
    private BookDBEJB database = null;

    public BookDB () throws Exception {
    }
    public void setBookId(String bookId) {
        this.bookId = bookId;
    }
    public void setDatabase(BookDBEJB database) {
        this.database = database;
    }
    public BookDetails getBookDetails()
        throws Exception {
        try {
            return (BookDetails)database.
                getBookDetails(bookId);
        } catch (BookNotFoundException ex) {
            throw ex;
        }
    }
    ...
}
```

Finally, this version of the example contains an applet to generate a dynamic digital clock in the banner. See Including an Applet (page 100) for a description of the JSP element that generates HTML for downloading the applet.

The source for the Duke's Bookstore application is located in the `<INSTALL>/j2eetutorial/examples/web/bookstore2` directory.

To deploy and run the example:

1. In the IDE, mount the filesystem
`<INSTALL>/j2eetutorial/examples/web/bookstore2`.
2. Expand the bookstore2 node.
3. Right-click the DigitalClock class and choose Compile.
4. Right-click the WEB-INF directory and choose Deploy.

5. Set up the PointBase database as described in Accessing Databases from Web Applications, page 39.
6. Open the bookstore URL `http://localhost:80/bookstore2/enter` in a browser.

To review the deployment settings:

1. Expand the WEB-INF node.
2. Select the `web.xml` file.
3. Select the Deployment property sheet.
4. Browse the listener property.
 - a. Click the Listeners property and open the property editor.
 - b. Note that the listener class is `listeners.ContextListener`.
5. Browse the JSP files definition and servlet mappings.
 - a. Click the JSP Files property and open the property editor.
 - b. Note the mappings between the JSP files and the URLs `/enter`, `/catalog`, `/bookdetails`, `/showcart`, `/cashier`, and `/receipt`.
6. Browse the resource references.
 - a. Select the Resources Property sheet.
 - a. Click the Resource References property and open the property editor.
 - b. Note the resource reference named `jdbc/BookDB`.

The Life Cycle of a JSP Page

A JSP page services requests as a servlet. Thus, the life cycle and many of the capabilities of JSP pages (in particular the dynamic aspects) are determined by Java Servlet technology, and much of the discussion in this chapter refers to functions described in Chapter 3.

When a request is mapped to a JSP page, it is handled by a special servlet that first checks whether the JSP page's servlet is older than the JSP page. If it is, it translates the JSP page into a servlet class and compiles the class. During development, one of the advantages of JSP pages over servlets is that the build process is performed automatically.

Translation and Compilation

During the translation phase each type of data in a JSP page is treated differently. Template data is transformed into code that will emit the data into the stream that returns data to the client. JSP elements are treated as follows:

- Directives are used to control how the Web container translates and executes the JSP page.
- Scripting elements are inserted into the JSP page's servlet class. See JSP Scripting Elements (page 95) for details.
- Elements of the form `<jsp:XXX ... />` are converted into method calls to JavaBeans components or invocations of the Java Servlet API.

For a JSP page named *pageName*, the source for a JSP page's servlet is kept in the file:

```
<S1AS7_HOME>/domains/domain1/server1/applications/j2ee-  
modules/context_root_n/pageName$.jsp.java
```

Both the translation and compilation phases can yield errors that are only observed when the page is requested for the first time. If an error occurs while the page is being translated (for example, if the translator encounters a malformed JSP element), the server will return a `ParseException`, and the servlet class source file will be empty or incomplete.

If an error occurs while the JSP page is being compiled (for example, there is a syntax error in a scriptlet), the server will return a `JasperException` and a message that includes the name of the JSP page's servlet and the line where the error occurred.

Once the page has been translated and compiled, the JSP page's servlet for the most part follows the servlet life cycle described in Servlet Life Cycle (page 51):

1. If an instance of the JSP page's servlet does not exist, the container
 - a. Loads the JSP page's servlet class
 - b. Instantiates an instance of the servlet class
 - c. Initializes the servlet instance by calling the `jspInit` method
2. The container invokes the `_jspService` method, passing a request and response object.

If the container needs to remove the JSP page's servlet, it calls the `jspDestroy` method.

Execution

You can control various JSP page execution parameters by using page directives. The directives that pertain to buffering output and handling errors are discussed here. Other directives are covered in the context of specific page authoring tasks throughout the chapter.

Buffering

When a JSP page is executed, output written to the response object is automatically buffered. You can set the size of the buffer with the following page directive:

```
<%@ page buffer="none|xxxkb" %>
```

A larger buffer allows more content to be written before anything is actually sent back to the client, thus providing the JSP page with more time to set appropriate status codes and headers or to forward to another Web resource. A smaller buffer decreases server memory load and allows the client to start receiving data more quickly.

Handling Errors

Any number of exceptions can arise when a JSP page is executed. To specify that the Web container should forward control to an error page if an exception occurs, include the following page directive at the beginning of your JSP page:

```
<%@ page errorPage="file_name" %>
```

The Duke's Bookstore application page `initdestroy.jsp` contains the directive

```
<%@ page errorPage="errorpage.jsp"%>
```

The beginning of `errorpage.jsp` indicates that it is serving as an error page with the following page directive:

```
<%@ page isErrorPage="true|false" %>
```

This directive makes the exception object (of type `javax.servlet.jsp.JspException`) available to the error page, so that you can retrieve, interpret, and possibly display information about the cause of the exception in the error page.

Note: You can also define error pages for the WAR that contains a JSP page. If error pages are defined for both the WAR and a JSP page, the JSP page's error page takes precedence.

Initializing and Finalizing a JSP Page

You can customize the initialization process to allow the JSP page to read persistent configuration data, initialize resources, and perform any other one-time activities by overriding the `jspInit` method of the `JspPage` interface. You release resources using the `jspDestroy` method. The methods are defined using JSP declarations, discussed in Declarations (page 95).

The bookstore example page `initdestroy.jsp` defines the `jspInit` method to retrieve the object database `BookDBAO` that accesses the bookstore database and stores a reference to the bean in `bookDBAO`.

```
private BookDBAO bookDBAO;
public void jspInit() {
    bookDBAO =
        (BookDBAO)getServletContext().getAttribute("bookDB");
    if (bookDBAO == null)
        System.out.println("Couldn't get database.");
}
```

When the JSP page is removed from service, the `jspDestroy` method releases the `BookDBAO` variable.

```
public void jspDestroy() {
    bookDBAO = null;
}
```

Since the enterprise bean is shared between all the JSP pages, it should be initialized when the application is started, instead of in each JSP page. Java Servlet technology provides application life-cycle events and listener classes for this purpose. As an exercise, you can move the code that manages the creation of the enterprise bean to a context listener class. See Handling Servlet Life Cycle Events (page 52) for the context listener that initializes the Java Servlet version of the bookstore application.

Creating Static Content

You create static content in a JSP page by simply writing it as if you were creating a page that consisted only of that content. Static content can be expressed in any text-based format, such as HTML, WML, and XML. The default format is HTML. If you want to use a format other than HTML, you include a page directive with the `contentType` attribute set to the format type at the beginning of your JSP page. For example, if you want a page to contain data expressed in the wireless markup language (WML), you need to include the following directive:

```
<%@ page contentType="text/vnd.wap.wml"%>
```

A registry of content type names is kept by the IANA at:

```
ftp://ftp.isi.edu/in-notes/iana/assignments/media-types
```

Creating Dynamic Content

You create dynamic content by accessing Java programming language objects from within scripting elements.

Using Objects within JSP Pages

You can access a variety of objects, including enterprise beans and JavaBeans components, within a JSP page. JSP technology automatically makes some objects available, and you can also create and access application-specific objects.

Implicit Objects

Implicit objects are created by the Web container and contain information related to a particular request, page, or application. Many of the objects are defined by

the Java Servlet technology underlying JSP technology and are discussed at length in Chapter 3. Table 4–2 summarizes the implicit objects.

Table 4–2 Implicit Objects

Variable	Class	Description
application	<code>javax.servlet.ServletContext</code>	The context for the JSP page's servlet and any Web components contained in the same application. See <i>Accessing the Web Context</i> (page 75).
config	<code>javax.servlet.ServletConfig</code>	Initialization information for the JSP page's servlet.
exception	<code>java.lang.Throwable</code>	Accessible only from an error page. See <i>Handling Errors</i> (page 90).
out	<code>javax.servlet.jsp.JspWriter</code>	The output stream.
page	<code>java.lang.Object</code>	The instance of the JSP page's servlet processing the current request. Not typically used by JSP page authors.
pageContext	<code>javax.servlet.jsp.PageContext</code>	The context for the JSP page. Provides a single API to manage the various scoped attributes described in <i>Using Scope Objects</i> (page 55). This API is used extensively when implementing tag handlers (see <i>Tag Handlers</i> , page 125).
request	subtype of <code>javax.servlet.HttpServletRequest</code>	The request triggering the execution of the JSP page. See <i>Getting Information from Requests</i> (page 60).
response	subtype of <code>javax.servlet.HttpServletResponse</code>	The response to be returned to the client. Not typically used by JSP page authors.
session	<code>javax.servlet.http.HttpSession</code>	The session object for the client. See <i>Maintaining Client State</i> (page 76).

Application-Specific Objects

When possible, application behavior should be encapsulated in objects so that page designers can focus on presentation issues. Objects can be created by devel-

opers who are proficient in the Java programming language and in accessing databases and other services. There are four ways to create and use objects within a JSP page:

- Instance and class variables of the JSP page's servlet class are created in *declarations* and accessed in *scriptlets* and *expressions*.
- Local variables of the JSP page's servlet class are created and used in *scriptlets* and *expressions*.
- Attributes of scope objects (see Using Scope Objects, page 55) are created and used in *scriptlets* and *expressions*.
- JavaBeans components can be created and accessed using streamlined JSP elements. These elements are discussed in JavaBeans Components in JSP Pages (page 103). You can also create a JavaBeans component in a declaration or scriptlet and invoke the methods of a JavaBeans component in a scriptlet or expression.

Declarations, scriptlets, and expressions are described in JSP Scripting Elements (page 95).

Shared Objects

The conditions affecting concurrent access to shared objects described in Controlling Concurrent Access to Shared Resources (page 56) apply to objects accessed from JSP pages that run as multithreaded servlets. You can indicate how a Web container should dispatch multiple client requests with the following page directive:

```
<%@ page isThreadSafe="true|false" %>
```

When `isThreadSafe` is set to `true`, the Web container may choose to dispatch multiple concurrent client requests to the JSP page. This is the *default* setting. If using `true`, you must ensure that you properly synchronize access to any shared objects defined at the page level. This includes objects created within declarations, JavaBeans components with page scope, and attributes of the page scope object.

If `isThreadSafe` is set to `false`, requests are dispatched one at a time, in the order they were received, and access to page level objects does not have to be controlled. However, you still must ensure that access to attributes of the application or session scope objects and to JavaBeans components with application or session scope is properly synchronized.

JSP Scripting Elements

JSP scripting elements are used to create and access objects, define methods, and manage the flow of control. Since one of the goals of JSP technology is to separate static template data from the code needed to dynamically generate content, very sparing use of JSP scripting is recommended. Much of the work that requires the use of scripts can be eliminated by using custom tags, described in Custom Tags in JSP Pages (page 113).

JSP technology allows a container to support any scripting language that can call Java objects. If you wish to use a scripting language other than the default, java, you must specify it in a page directive at the beginning of a JSP page:

```
<%@ page language="scripting language" %>
```

Since scripting elements are converted to programming language statements in the JSP page's servlet class, you must import any classes and packages used by a JSP page. If the page language is java, you import a class or package with the page directive:

```
<%@ page import="packagename.*, fully_qualified_classname" %>
```

For example, the bookstore example page `showcart.jsp` imports the classes needed to implement the shopping cart with the following directive:

```
<%@ page import="java.util.*, cart.*" %>
```

Declarations

A *JSP declaration* is used to declare variables and methods in a page's scripting language. The syntax for a declaration is as follows:

```
<%! scripting language declaration %>
```

When the scripting language is the Java programming language, variables and methods in JSP declarations become declarations in the JSP page's servlet class.

The bookstore example page `initdestroy.jsp` defines an instance variable named `bookDBAO` and the initialization and finalization methods `jspInit` and `jspDestroy` discussed earlier in a declaration:

```
<%!
    private BookDBAO bookDBAO;

    public void jspInit() {
        ...
    }
    public void jspDestroy() {
        ...
    }
%>
```

Scriptlets

A *JSP scriptlet* is used to contain any code fragment that is valid for the scripting language used in a page. The syntax for a scriptlet is as follows:

```
<%
    scripting language statements
%>
```

When the scripting language is set to `java`, a scriptlet is transformed into a Java programming language statement fragment and is inserted into the service method of the JSP page's servlet. A programming language variable created within a scriptlet is accessible from anywhere within the JSP page.

The JSP page `showcart.jsp` contains a scriptlet that retrieves an iterator from the collection of items maintained by a shopping cart and sets up a construct to loop through all the items in the cart. Inside the loop, the JSP page extracts properties of the book objects and formats them using HTML markup. Since the `while` loop opens a block, the HTML markup is followed by a scriptlet that closes the block.

```
<%
    Iterator i = cart.getItems().iterator();
    while (i.hasNext()) {
        ShoppingCartItem item =
            (ShoppingCartItem)i.next();
        BookDetails bd = (BookDetails)item.getItem();
%>

    <tr>
```

```

        <td align="right" bgcolor="#ffffff">
            <%=item.getQuantity()%>
        </td>
        <td bgcolor="#ffffaa">
            <strong><a href="
            <%=request.getContextPath()%>/bookdetails?bookId=
            <%=bd.getBookId()%>"><%=bd.getTitle()%></a></strong>
        </td>
        ...
    <%
        // End of while
    }
%>

```

The result of executing the page appears in Figure 4–2.

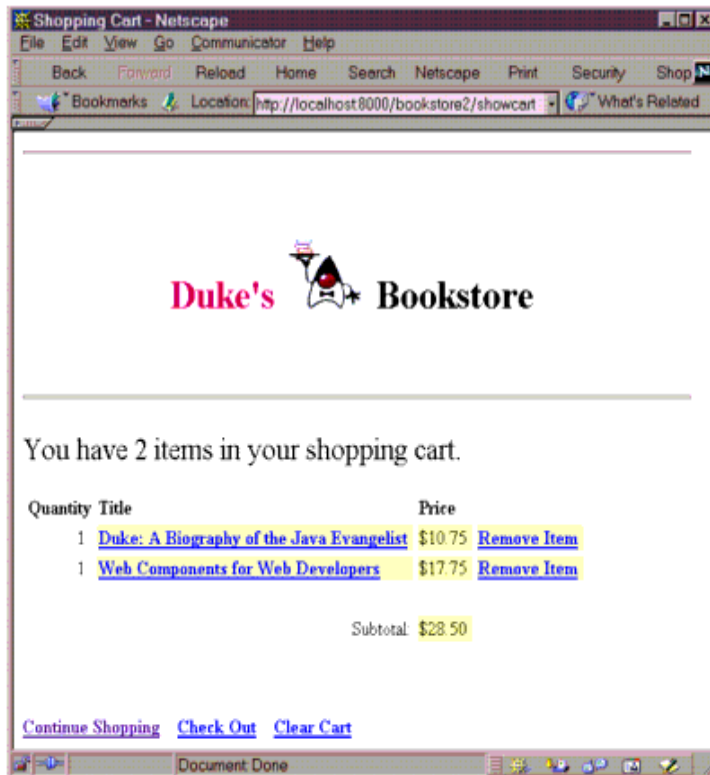


Figure 4–2 Duke's Bookstore Shopping Cart

Expressions

A *JSP expression* is used to insert the value of a scripting language expression, converted into a string, into the data stream returned to the client. When the scripting language is the Java programming language, an expression is transformed into a statement that converts the value of the expression into a `String` object and inserts it into the implicit out object.

The syntax for an expression is as follows:

```
<%= scripting language expression %>
```

Note that a semicolon is not allowed within a JSP expression, even if the same expression has a semicolon when you use it within a scriptlet.

The following scriptlet retrieves the number of items in a shopping cart:

```
<%  
    // Print a summary of the shopping cart  
    int num = cart.getNumberOfItems();  
    if (num > 0) {  
%>
```

Expressions are then used to insert the value of `num` into the output stream and determine the appropriate string to include after the number:

```
<font size="+2">  
<%=messages.getString("CartContents")%> <%=num%>  
    <%= (num==1 ? <%=messages.getString("CartItem")%> :  
    <%=messages.getString("CartItems"))%></font>
```

Including Content in a JSP Page

There are two mechanisms for including another Web resource in a JSP page: the `include` directive and the `jsp:include` element.

The `include` directive is processed when the JSP page is *translated* into a servlet class. The effect of the directive is to insert the text contained in another file—either static content or another JSP page—in the including JSP page. You would probably use the `include` directive to include banner content, copyright infor-

mation, or any chunk of content that you might want to reuse in another page. The syntax for the `include` directive is as follows:

```
<%@ include file="filename" %>
```

For example, all the bookstore application pages include the file `banner.jsp` which contains the banner content, with the following directive:

```
<%@ include file="banner.jsp" %>
```

In addition, the pages `bookstore.jsp`, `bookdetails.jsp`, `catalog.jsp`, and `showcart.jsp` include JSP elements that create and destroy a database bean with the following directive:

```
<%@ include file="initdestroy.jsp" %>
```

Because you must statically put an `include` directive in each file that reuses the resource referenced by the directive, this approach has its limitations. For a more flexible approach to building pages out of content chunks, see *A Template Tag Library* (page 141).

The `jsp:include` element is processed when a JSP page is *executed*. The `include` action allows you to include either a static or dynamic resource in a JSP file. The results of including static and dynamic resources are quite different. If the resource is static, its content is inserted into the calling JSP file. If the resource is dynamic, the request is sent to the included resource, the included page is executed, and then the result is included in the response from the calling JSP page. The syntax for the `jsp:include` element is:

```
<jsp:include page="includedPage" />
```

The date application introduced at the beginning of this chapter includes the page that generates the display of the localized date with the following statement:

```
<jsp:include page="date.jsp"/>
```

Transferring Control to Another Web Component

The mechanism for transferring control to another Web component from a JSP page uses the functionality provided by the Java Servlet API as described in Transferring Control to Another Web Component (page 74). You access this functionality from a JSP page with the `jsp:forward` element:

```
<jsp:forward page="/main.jsp" />
```

Note that if any data has already been returned to a client, the `jsp:forward` element will fail with an `IllegalStateException`.

jsp:param Element

When an `include` or `forward` element is invoked, the original request object is provided to the target page. If you wish to provide additional data to that page, you can append parameters to the request object with the `jsp:param` element:

```
<jsp:include page="..." >  
  <jsp:param name="param1" value="value1"/>  
</jsp:include>
```

Including an Applet

You can include an applet or JavaBeans component in a JSP page by using the `jsp:plugin` element. This element generates HTML that contains the appropriate client-browser-dependent constructs (`<object>` or `<embed>`) that will result in the download of the Java Plug-in software (if required) and client-side component and subsequent execution of any client-side component. The syntax for the `jsp:plugin` element is as follows:

```
<jsp:plugin  
  type="bean|applet"  
  code="objectCode"  
  codebase="objectCodebase"  
  { align="alignment" }  
  { archive="archiveList" }  
  { height="height" }  
  { hspace="hspace" }
```



```
{ jreversion="jreversion" }  
{ name="componentName" }  
{ vspace="vspace" }  
{ width="width" }  
{ nspluginurl="url" }  
{ iepluginurl="url" } >  
{ <jsp:params>  
  { <jsp:param name="paramName" value= paramValue" /> }+  
</jsp:params> }  
{ <jsp:fallback> arbitrary_text </jsp:fallback> }  
</jsp:plugin>
```

The `jsp:plugin` tag is replaced by either an `<object>` or `<embed>` tag as appropriate for the requesting client. The attributes of the `jsp:plugin` tag provide configuration data for the presentation of the element as well as the version of the plug-in required. The `nspluginurl` and `iepluginurl` attributes specify the URL where the plug-in can be downloaded.

The `jsp:param` elements specify parameters to the applet or JavaBeans component. The `jsp:fallback` element indicates the content to be used by the client browser if the plug-in cannot be started (either because `<object>` or `<embed>` is not supported by the client or because of some other problem).

If the plug-in can start but the applet or JavaBeans component cannot be found or started, a plug-in-specific message will be presented to the user, most likely a pop-up window reporting a `ClassNotFoundException`. If the applet or JavaBeans component cannot be found, most likely it has been packaged incorrectly. Client-side classes such as applets must be packaged in the root of the Web module, *not* in the `WEB-INF/classes` directory. So, the `DigitalClock` applet contained in the `bookstore2` application is packaged at the same level as the JSP pages.

The Duke's Bookstore page `banner.jsp` that creates the banner displays a dynamic digital clock generated by `DigitalClock`:

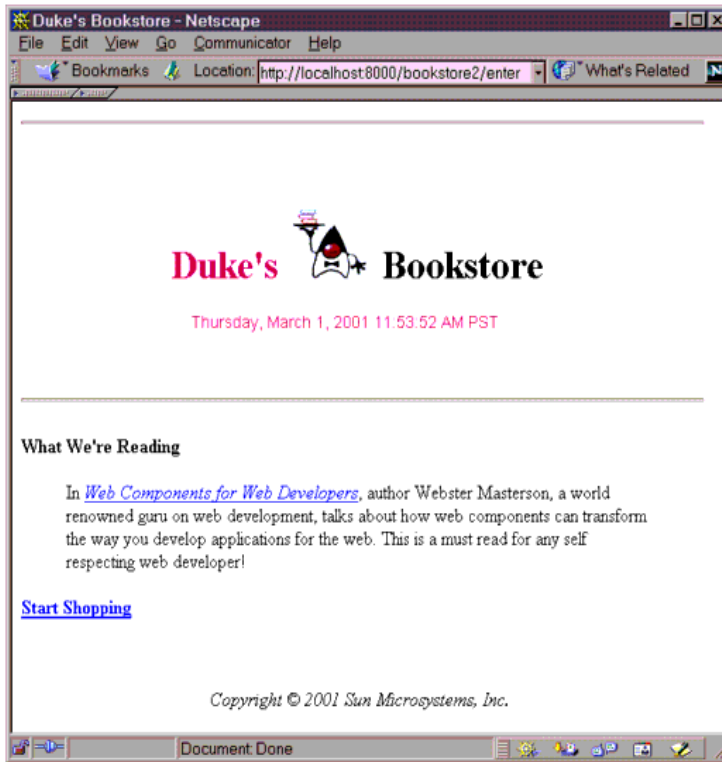


Figure 4-3 Duke's Bookstore with Applet

The `jsp:plugin` element used to download the applet follows:

```
<jsp:plugin
  type="applet"
  code="DigitalClock.class"
  codebase="/bookstore2"
  jreversion="1.4"
  align="center" height="25" width="300"
<jsp:plugin type="applet" code="DigitalClock.class"
codebase="/bookstore2" jreversion="1.4" align="center"
height="25" width="300"
nspluginurl="http://java.sun.com/j2se/1.4/download.html"
iepluginurl="http://java.sun.com/j2se/1.4/download.html" >
<jsp:params>
```

```
<jsp:param name="language"
  value="<%=request.getLocale().getLanguage()%>" />
<jsp:param name="country"
  value="<%=request.getLocale().getCountry()%>" />
<jsp:param name="bgcolor" value="FFFFFF" />
<jsp:param name="fgcolor" value="CC0066" />
</jsp:params>
<jsp:fallback>
  <p>Unable to start plugin.</p>
</jsp:fallback>
</jsp:plugin>
```

JavaBeans Components in JSP Pages

JavaBeans components are Java classes that can be easily reused and composed together into applications. Any Java class that follows certain design conventions can be a JavaBeans component.

JavaServer Pages technology directly supports using JavaBeans components with JSP language elements. You can easily create and initialize beans and get and set the values of their properties. This chapter provides basic information about JavaBeans components and the JSP language elements for accessing JavaBeans components in your JSP pages. For further information about the JavaBeans component model see <http://java.sun.com/products/javabeans>.

JavaBeans Component Design Conventions

JavaBeans component design conventions govern the properties of the class and govern the public methods that give access to the properties.

A JavaBeans component property can be

- Read/write, read-only, or write-only
- Simple, which means it contains a single value, or indexed, which means it represents an array of values

There is no requirement that a property be implemented by an instance variable; the property must simply be accessible using public methods that conform to certain conventions:

- For each readable property, the bean must have a method of the form

```
PropertyClass getProperty() { ... }
```

- For each writable property, the bean must have a method of the form

```
setProperty(PropertyClass pc) { ... }
```

In addition to the property methods, a JavaBeans component must define a constructor that takes no parameters.

The Duke's Bookstore application JSP pages `enter.jsp`, `bookdetails.jsp`, `catalog.jsp`, and `showcart.jsp` use the `database.BookDB` and `database.BookDetails` JavaBeans components. `BookDB` provides a JavaBeans component front end to the access object `BookDBAO`. Both beans are used extensively by bean-oriented custom tags (see Custom Tags in JSP Pages, page 113). The JSP pages `showcart.jsp` and `cashier.jsp` use `cart.ShoppingCart` to represent a user's shopping cart.

The JSP pages `catalog.jsp`, `showcart.jsp`, and `cashier.jsp` use the `util.Currency` JavaBeans component to format currency in a locale-sensitive manner. The bean has two writable properties, `locale` and `amount`, and one readable property, `format`. The `format` property does not correspond to any instance variable, but returns a function of the `locale` and `amount` properties.

```
public class Currency {
    private Locale locale;
    private double amount;
    public Currency() {
        locale = null;
        amount = 0.0;
    }
    public void setLocale(Locale l) {
        locale = l;
    }
    public void setAmount(double a) {
        amount = a;
    }
    public String getFormat() {
        NumberFormat nf =
```

```
        NumberFormat.getCurrencyInstance(locale);
    return nf.format(amount);
    }
}
```

Why Use a JavaBeans Component?

A JSP page can create and use any type of Java programming language object within a declaration or scriptlet. The following scriptlet creates the bookstore shopping cart and stores it as a session attribute:

```
<%
    ShoppingCart cart = (ShoppingCart)session.
        getAttribute("cart");
    // If the user has no cart, create a new one
    if (cart == null) {
        cart = new ShoppingCart();
        session.setAttribute("cart", cart);
    }
%>
```

If the shopping cart object conforms to JavaBeans conventions, JSP pages can use JSP elements to create and access the object. For example, the Duke's Bookstore pages `bookdetails.jsp`, `catalog.jsp`, and `showcart.jsp` replace the scriptlet with the much more concise JSP `useBean` element:

```
<jsp:useBean id="cart" class="cart.ShoppingCart"
    scope="session"/>
```

Creating and Using a JavaBeans Component

You declare that your JSP page will use a JavaBeans component using either one of the following formats:

```
<jsp:useBean id="beanName"
    class="fully_qualified_classname" scope="scope"/>
```

or

```
<jsp:useBean id="beanName"  
  class="fully_qualified_classname" scope="scope">  
  <jsp:setProperty .../>  
</jsp:useBean>
```

The second format is used when you want to include `jsp:setProperty` statements, described in the next section, for initializing bean properties.

The `jsp:useBean` element declares that the page will use a bean that is stored within and accessible from the specified scope, which can be `application`, `session`, `request`, or `page`. If no such bean exists, the statement creates the bean and stores it as an attribute of the scope object (see *Using Scope Objects* (page 55)). The value of the `id` attribute determines the *name* of the bean in the scope and the *identifier* used to reference the bean in other JSP elements and scriptlets.

Note: In *JSP Scripting Elements* (page 95), we mentioned that you must import any classes and packages used by a JSP page. This rule is slightly altered if the class is only referenced by `useBean` elements. In these cases, you must only import the class if the class is in the unnamed package. For example, in *What Is a JSP Page?* (page 83), the page `index.jsp` imports the `MyLocales` class. However, in the *Duke's Bookstore* example, all classes are contained in packages and thus are not explicitly imported.

The following element creates an instance of `Currency` if none exists, stores it as an attribute of the `session` object, and makes the bean available throughout the session by the identifier `currency`:

```
<jsp:useBean id="currency" class="util.Currency"  
  scope="session"/>
```

Setting JavaBeans Component Properties

There are two ways to set JavaBeans component properties in a JSP page: with the `jsp:setProperty` element or with a scriptlet

```
<% beanName.setPropName(value); %>
```

The syntax of the `jsp:setProperty` element depends on the source of the property value. Table 4–3 summarizes the various ways to set a property of a JavaBeans component using the `jsp:setProperty` element.

Table 4–3 Setting JavaBeans Component Properties

Value Source	Element Syntax
String constant	<code><jsp:setProperty name="beanName" property="propName" value="string constant"/></code>
Request parameter	<code><jsp:setProperty name="beanName" property="propName" param="paramName"/></code>
Request parameter name matches bean property	<code><jsp:setProperty name="beanName" property="propName"/></code> <code><jsp:setProperty name="beanName" property="*/></code>
Expression	<code><jsp:setProperty name="beanName" property="propName" value="<%= expression %>/></code>
	<ol style="list-style-type: none"> 1. <i>beanName</i> must be the same as that specified for the <i>id</i> attribute in a <i>useBean</i> element. 2. There must be a <i>setPropName</i> method in the JavaBeans component. 3. <i>paramName</i> must be a request parameter name.

A property set from a constant string or request parameter must have a type listed in Table 4–4. Since both a constant and request parameter are strings, the Web container automatically converts the value to the property's type; the conversion applied is shown in the table. String values can be used to assign values to a property that has a `PropertyEditor` class. When that is the case, the `setAsText(String)` method is used. A conversion failure arises if the method throws

an `IllegalArgumentException`. The value assigned to an indexed property must be an array, and the rules just described apply to the elements.

Table 4-4 Valid Value Assignments

Property Type	Conversion on String Value
Bean Property	Uses <code>setAsText(<i>string-literal</i>)</code>
<code>boolean</code> or <code>Boolean</code>	As indicated in <code>java.lang.Boolean.valueOf(String)</code>
<code>byte</code> or <code>Byte</code>	As indicated in <code>java.lang.Byte.valueOf(String)</code>
<code>char</code> or <code>Character</code>	As indicated in <code>java.lang.String.charAt(0)</code>
<code>double</code> or <code>Double</code>	As indicated in <code>java.lang.Double.valueOf(String)</code>
<code>int</code> or <code>Integer</code>	As indicated in <code>java.lang.Integer.valueOf(String)</code>
<code>float</code> or <code>Float</code>	As indicated in <code>java.lang.Float.valueOf(String)</code>
<code>long</code> or <code>Long</code>	As indicated in <code>java.lang.Long.valueOf(String)</code>
<code>short</code> or <code>Short</code>	As indicated in <code>java.lang.Short.valueOf(String)</code>
<code>Object</code>	<code>new String(<i>string-literal</i>)</code>

You would use a runtime expression to set the value of a property whose type is a compound Java programming language type. Recall from Expressions (page 98) that a JSP expression is used to insert the value of a scripting language expression, converted into a `String`, into the stream returned to the client. When used within a `setProperty` element, an expression simply returns its value; no automatic conversion is performed. As a consequence, the type returned from an expression must match or be castable to the type of the property.

The Duke's Bookstore application demonstrates how to use the `setProperty` element and a scriptlet to set the current book for the database helper bean. For example, `bookstore3/web/bookdetails.jsp` uses the form:

```
<jsp:setProperty name="bookDB" property="bookId"/>
```


whereas `bookstore2/web/bookdetails.jsp` uses the form:

```
<% bookDB.setBookId(bookId); %>
```

The following fragments from the page `bookstore3/web/showcart.jsp` illustrate how to initialize a currency bean with a `Locale` object and amount determined by evaluating request-time expressions. Because the first initialization is nested in a `useBean` element, it is only executed when the bean is created.

```
<jsp:useBean id="currency" class="util.Currency"
  scope="session">
  <jsp:setProperty name="currency" property="locale"
    value="<%= request.getLocale() %>"/>
</jsp:useBean>

<jsp:setProperty name="currency" property="amount"
  value="<%= cart.getTotal() %>"/>
```

Retrieving JavaBeans Component Properties

There are several ways to retrieve JavaBeans component properties. Two of the methods (the `jsp:getProperty` element and an expression) convert the value of the property into a `String` and insert the value into the current implicit out object:

- `<jsp:getProperty name="beanName" property="propName"/>`
- `<%= beanName.getPropName() %>`

For both methods, *beanName* must be the same as that specified for the `id` attribute in a `useBean` element, and there must be a `getPropName` method in the JavaBeans component.

If you need to retrieve the value of a property without converting it and inserting it into the out object, you must use a scriptlet:

```
<% Object o = beanName.getPropName(); %>
```

Note the differences between the expression and the scriptlet; the expression has an `=` after the opening `%` and does not terminate with a semicolon, as does the scriptlet.

The Duke's Bookstore application demonstrates how to use both forms to retrieve the formatted currency from the currency bean and insert it into the page. For example, `bookstore3/web/showcart.jsp` uses the form

```
<jsp:getProperty name="currency" property="format"/>
```

whereas `bookstore2/web/showcart.jsp` uses the form:

```
<%= currency.getFormat() %>
```

The Duke's Bookstore application page `bookstore2/web/showcart.jsp` uses the following scriptlet to retrieve the number of books from the shopping cart bean and open a conditional insertion of text into the output stream:

```
<%  
    // Print a summary of the shopping cart  
    int num = cart.getNumberOfItems();  
    if (num > 0) {  
%>
```

Although scriptlets are very useful for dynamic processing, using custom tags (see Custom Tags in JSP Pages, page 113) to access object properties and perform flow control is considered to be a better approach. For example, `bookstore3/web/showcart.jsp` replaces the scriptlet with the following custom tags:

```
<bean:define id="num" name="cart" property="numberOfItems" />  
<logic:greaterThan name="num" value="0" >
```

Figure 4-4 summarizes where various types of objects are stored and how those objects can be accessed from a JSP page. Objects created by the `jsp:useBean` tag are stored as attributes of the scope objects and can be accessed by `jsp:[get|set]Property` tags and in scriptlets and expressions. Objects created

in declarations and scriptlets are stored as variables of the JSP page's servlet class and can be accessed in scriptlets and expressions.

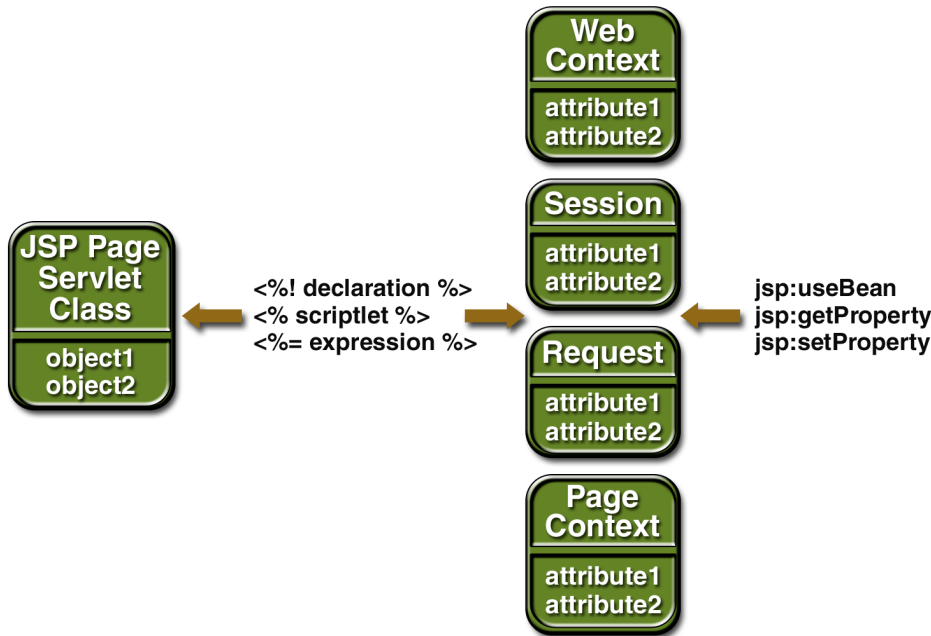


Figure 4-4 Accessing Objects From a JSP Page

Extending the JSP Language

You can perform a wide variety of dynamic processing tasks, including accessing databases, using enterprise services such as e-mail and directories, and flow control, with JavaBeans components in conjunction with scriptlets. One of the drawbacks of scriptlets, however, is that they tend to make JSP pages more difficult to maintain. Alternatively, JSP technology provides a mechanism, called *custom tags*, that allows you to encapsulate dynamic functionality in objects that are accessed through extensions to the JSP language. Custom tags bring the benefits of another level of componentization to JSP pages.

For example, recall the scriptlet used to loop through and display the contents of the Duke's Bookstore shopping cart:

```
<%
    Iterator i = cart.getItems().iterator();
    while (i.hasNext()) {
        ShoppingCartItem item =
            (ShoppingCartItem)i.next();
        ...
    }
%>
```

An `iterate` custom tag eliminates the code logic and manages the scripting variable `item` that references elements in the shopping cart:

```
<logic:iterate id="item"
    collection="<%=cart.getItems()%>">
    <tr>
    <td align="right" bgcolor="#ffffff">
    <%=item.getQuantity()%>
    </td>
    ...
</logic:iterate>
```

Custom tags are packaged and distributed in a unit called a *tag library*. The syntax of custom tags is the same as that used for the JSP elements, namely `<prefix:tag>`, for custom tags, however, `prefix` is defined by the *user* of the tag library, and `tag` is defined by the *tag developer*. Custom Tags in JSP Pages (page 113) explains how to use and develop custom tags.

Further Information

For further information on JavaServer Pages technology see:

- Resources listed on the Web site <http://java.sun.com/products/jsp>.
- The JavaServer Pages 1.2 Specification.

Custom Tags in JSP Pages

Stephanie Bodoff

THE standard JSP tags for invoking operations on JavaBeans components and performing request dispatching simplify JSP page development and maintenance. JSP technology also provides a mechanism for encapsulating other types of dynamic functionality in *custom tags*, which are extensions to the JSP language. Custom tags are usually distributed in the form of a *tag library*, which defines a set of related custom tags and contains the objects that implement the tags.

Some examples of tasks that can be performed by custom tags include operations on implicit objects, processing forms, accessing databases and other enterprise services such as e-mail and directories, and performing flow control. JSP tag libraries are created by developers who are proficient at the Java programming language and expert in accessing data and other services, and are used by Web application designers who can focus on presentation issues rather than being concerned with how to access enterprise services. As well as encouraging division of labor between library developers and library users, custom tags increase productivity by encapsulating recurring tasks so that they can be reused across more than one application.

Tag libraries are receiving a great deal of attention in the JSP technology community. For more information about tag libraries and for pointers to some freely-available libraries, see

<http://java.sun.com/products/jsp/taglibraries.html>

What Is a Custom Tag?

A custom tag is a user-defined JSP language element. When a JSP page containing a custom tag is translated into a servlet, the tag is converted to operations on an object called a *tag handler*. The Web container then invokes those operations when the JSP page's servlet is executed.

Custom tags have a rich set of features. They can

- Be customized via attributes passed from the calling page.
- Access all the objects available to JSP pages.
- Modify the response generated by the calling page.
- Communicate with each other. You can create and initialize a JavaBeans component, create a variable that refers to that bean in one tag, and then use the bean in another tag.
- Be nested within one another, allowing for complex interactions within a JSP page.

The Example JSP Pages

This chapter describes the tasks involved in using and defining tags. The chapter illustrates the tasks with excerpts from the JSP version of the Duke's Bookstore application discussed in The Example JSP Pages (page 86) rewritten to take advantage of two tag libraries: Struts and template. The third section in the chapter, Examples (page 137), describes two tags in detail: an iterator tag and the set of tags in the template tag library.

The Struts tag library provides a framework for building internationalized Web applications that implement the Model-View-Controller design pattern. Struts includes a comprehensive set of utility custom tags for handling:

- HTML forms
- Templates
- JavaBeans components
- Logic processing

The Duke's Bookstore application uses tags from the Struts bean and logic sublibraries.

The template tag library defines a set of tags for creating an application template. The template is a JSP page with placeholders for the parts that need to change with each screen. Each of these placeholders is referred to as a parameter of the template. For example, a simple template could include a title parameter for the top of the generated screen and a body parameter to refer to a JSP page for the custom content of the screen. The template is created with a set of nested tags—`definition`, `screen`, and `parameter`—that are used to build a table of screen definitions for Duke's Bookstore and with an `insert` tag to insert parameters from the table into the screen.

Figure 5–1 shows the flow of a request through the following Duke's Bookstore Web components:

- `template.jsp`, which determines the structure of each screen. It uses the `insert` tag to compose a screen from subcomponents.
- `screendefinitions.jsp`, which defines the subcomponents used by each screen. All screens have the same banner, but different title and body content (specified by the JSP Pages column in Table 4–1).
- Dispatcher, a servlet, which processes requests and forwards to `template.jsp`.

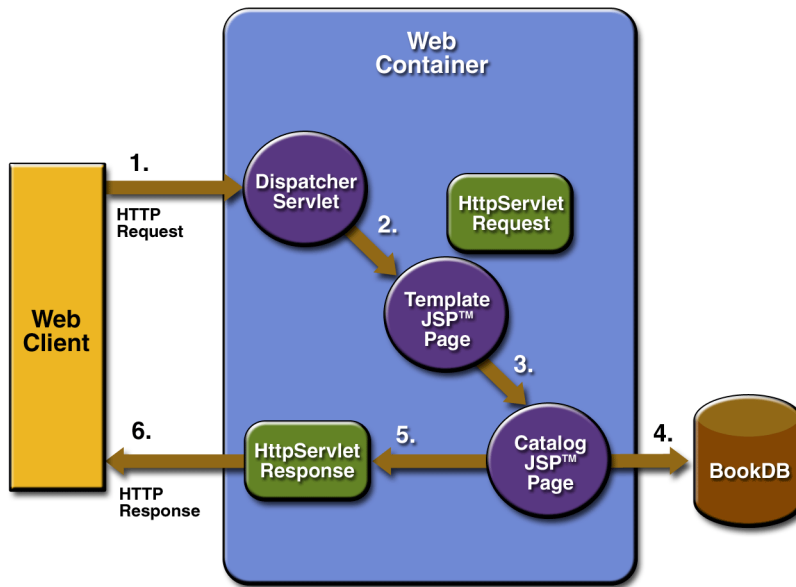


Figure 5–1 Request Flow Through Duke's Bookstore Components

The template tag library files are located in the directory `<INSTALL>/j2eetutorial/examples/web/template`. The classes are packaged into a JAR and already included in the `WEB-INF/lib` directory of all successive versions of the Duke's Bookstore example. To recreate the template library JAR:

1. In the IDE, mount the filesystem `<INSTALL>/j2eetutorial/examples/web/template`.
2. Right-click the `template` node and choose `Compile`.
3. Expand the `template` node.
4. Create the `template` JAR.
 - a. Right-click the `template` TLD and choose `Create Tag Library JAR`.
 - b. Right-click the `template` JAR recipe node and choose `Compile`.
5. Expand the `template` JAR recipe node.

The source for the Duke's Bookstore application is located in the `<INSTALL>/j2eetutorial/examples/web/bookstore3` directory.

To deploy and run the example:

1. In the IDE, mount the filesystem
`<INSTALL>/j2eetutorial/examples/web/bookstore3`.
2. Expand the bookstore3 node.
3. Right-click the WEB-INF directory and choose Deploy.
4. Set up the PointBase database as described in Accessing Databases from Web Applications, page 39.
5. Open the bookstore URL `http://localhost:80/bookstore3/enter`.

To review the deployment settings:

1. Expand the WEB-INF node.
2. Select the `web.xml` file.
3. Select the Deployment property sheet.
4. Browse the listener property.
 - a. Click the Listeners property and open the property editor.
 - b. Notice that the listener class is `listeners.ContextListener`.
5. Browse the servlet definition and servlet mappings.
 - a. Click the Servlets property and open the property editor.
 - b. Notice that the Dispatcher servlet is implemented by the Dispatcher class and that the URLs `/enter`, `/catalog`, `/bookdetails`, `/showcart`, `/cashier`, and `/receipt` are mapped to the Dispatcher servlet.
6. Browse the tag libraries mapping.
 - a. Click the Tag Libraries property and open the property editor.
 - b. Notice that the relative URI `/struts` is mapped to `/WEB-INF/lib/struts.jar`.
 - c. Notice that the relative URI `/template` is mapped to `/WEB-INF/lib/template.jar`.
7. Browse the resource references.
 - a. Select the Resources Property sheet.
 - a. Click the Resource References property and open the property editor.
 - b. Note the resource reference named `jdbc/BookDB`.

Using Tags

This section describes how a JSP page uses tags and introduces the different types of tags.

To use a tag, a page author must do two things:

- Declare the tag library containing the tag
- Make the tag library implementation available to the Web application

Declaring Tag Libraries

You declare that a JSP page will use tags defined in a tag library by including a `taglib` directive in the page before any custom tag is used:

```
<%@ taglib uri="/WEB-INF/template.tld" prefix="tt" %>
```

The `uri` attribute refers to a URI that uniquely identifies the tag library descriptor (TLD), described in Tag Library Descriptors (page 122). This URI can be direct or indirect. The `prefix` attribute defines the prefix that distinguishes tags defined by a given tag library from those provided by other tag libraries.

Tag library descriptor file names must have the extension `.tld`. TLD files are stored in the `WEB-INF` directory of the WAR or in a subdirectory of `WEB-INF`. You can reference a TLD directly and indirectly.

The following `taglib` directive directly references a TLD filename:

```
<%@ taglib uri="/WEB-INF/template.tld" prefix="tt" %>
```

You can also directory reference a TLD by referencing the library in which the TLD is contained:

```
<%@ taglib uri="/WEB-INF/lib/template.jar" prefix="tt" %>
```

This `taglib` directive uses a short logical name to indirectly reference the TLD:

```
<%@ taglib uri="/template" prefix="tt" %>
```

You map a logical name to an absolute location in the Web application deployment descriptor. To map the logical name `/template` to the absolute location `/WEB-INF/template.tld`, you add a tag library element to `web.xml`.

You can also reference a TLD in a `taglib` directive with an absolute URI:

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
```

This directive references the JSTL core tag library discussed in Chapter 6. When you use an absolute URI, you do not have to add the `taglib` element to `web.xml`; the JSP container automatically locates the TLD inside the JSTL library implementation.

Making the Tag Library Implementation Available

A tag library implementation can be made available to a Web application in two basic ways. The classes implementing the tag handlers can be stored in an unpacked form in the `WEB-INF/classes` subdirectory of the Web application. Alternatively, if the library is distributed as a JAR, it is stored in the `WEB-INF/lib` directory of the Web application.

Types of Tags

JSP custom tags are written using XML syntax. They have a start tag and end tag, and possibly a body:

```
<tt:tag>
  body
</tt:tag>
```

A custom tag with no body is expressed as follows:

```
<tt:tag />
```

Simple Tags

A simple tag contains no body and no attributes:

```
<tt:simple />
```

Tags with Attributes

A custom tag can have attributes. Attributes are listed in the start tag and have the syntax `attr="value"`. Attribute values serve to customize the behavior of a custom tag just as parameters are used to customize the behavior of a method. You specify the types of a tag's attributes in a tag library descriptor, (see Tags with Attributes, page 127).

You can set an attribute value from a `String` constant or a runtime expression. The conversion process between the constants and runtime expressions and attribute types follows the rules described for JavaBeans component properties in Setting JavaBeans Component Properties (page 106).

The attributes of the Struts `logic:present` tag determine whether the body of the tag is evaluated. In the following example, an attribute specifies a request parameter named `Clear`:

```
<logic:present parameter="Clear">
```

The Duke's Bookstore application page `catalog.jsp` uses a runtime expression to set the value of the attribute that determines the collection of books over which the Struts `logic:iterate` tag iterates:

```
<logic:iterate collection="<%=bookDB.getBooks()%>"  
  id="book" type="database.BookDetails">
```

Tags with Bodies

A custom tag can contain custom and core tags, scripting elements, HTML text, and tag-dependent body content between the start and end tag.

In the following example, the Duke's Bookstore application page `showcart.jsp` uses the Struts `logic:present` tag to clear the shopping cart and print a message if the request contains a parameter named `Clear`:

```
<logic:present parameter="Clear">  
  <% cart.clear(); %>  
  <font color="#ff0000" size="+2"><strong>  
    You just cleared your shopping cart!  
  </strong><br>&nbsp;<br></font>  
</logic:present>
```

Choosing between Passing Information as Attributes or Body

As shown in the last two sections, it is possible to pass a given piece of data as an attribute of the tag or as the tag's body. Generally speaking, any data that is a simple string or can be generated by evaluating a simple expression is best passed as an attribute.

Tags That Define Scripting Variables

A custom tag can define a variable that can be used in scripts within a page. The following example illustrates how to define and use a scripting variable that contains an object returned from a JNDI lookup. Examples of such objects include enterprise beans, transactions, databases, environment entries, and so on:

```
<tt:lookup id="tx" type="UserTransaction"
  name="java:comp/UserTransaction" />
<% tx.begin(); %>
```

In the Duke's Bookstore application, several pages use bean-oriented tags from Struts to define scripting variables. For example, `bookdetails.jsp` uses the `bean:parameter` tag to create the `bookId` scripting variable and set it to the value of the `bookId` request parameter. The `jsp:setProperty` statement also sets the `bookId` property of the `bookDB` object to the value of the `bookId` request parameter. The `bean:define` tag retrieves the value of the bookstore database property `bookDetails` and defines the result as the scripting variable `book`:

```
<bean:parameter id="bookId" name="bookId" />
<jsp:setProperty name="bookDB" property="bookId"/>
<bean:define id="book" name="bookDB" property="bookDetails"
  type="database.BookDetails"/>
<h2><jsp:getProperty name="book" property="title"></h2>
```

Finally, the `iterator` tag described in detail later in the chapter sets a scripting variable `name` to each member of a collection:

```
<tl:iterator name="member" type="org.Member"
  group="<%= dept.getMembers() %>">
```

Cooperating Tags

Custom tags can cooperate with each other through shared objects.

In the following example, `tag1` creates an object called `obj1`, which is then reused by `tag2`.

```
<tt:tag1 attr1="obj1" value1="value" />
<tt:tag2 attr1="obj1" />
```

In the next example, an object created by the enclosing tag of a group of nested tags is available to all inner tags. Since the object is not named, the potential for naming conflicts is reduced. This example illustrates how a set of cooperating nested tags would appear in a JSP page.

```
<tt:outerTag>
  <tt:innerTag />
</tt:outerTag>
```

The Duke's Bookstore page `template.jsp` uses a set of cooperating tags to define the screens of the application. These tags are described in A Template Tag Library (page 141).

Defining Tags

To define a tag, you need to:

- Declare the tag in a tag library descriptor
- Develop a tag handler and helper classes for the tag

This section describes the properties of TLDs and tag handlers and explains how to develop library descriptor elements and tag handlers for each type of tag introduced in the previous section.

Tag Library Descriptors

A *tag library descriptor* (TLD) is an XML document that describes a tag library. A TLD contains information about a library as a whole and about each tag contained in the library. TLDs are used by a Web container to validate the tags and by JSP page development tools.

When you create an tag library in the IDE, it creates a TLD file. You customize a tag library by right-clicking the TLD and choosing *customize*. The code generation properties do not appear in the TLD, but simply determine the location of

generated tag handlers and the tag library JAR. The properties of the tag library are listed in Table 5–1:

Table 5–1 Tag Library Properties

Property	Description
Version	The tag library's version
JSP Version	The JSP specification version that the tag library requires
Short name	Optional name that could be used by a JSP page authoring tool to create names with a mnemonic value
URI	A URI that uniquely identifies the tag library
Display name	Optional name intended to be displayed by tools
Small Icon	Optional small-icon that can be used by tools
Large Icon	Optional large-icon that can be used by tools
Description	Optional tag-specific information

listener Element

A tag library can specify some classes that are event listeners (see Handling Servlet Life Cycle Events, page 52). The listeners are listed in the TLD as `listener` elements, and the Web container will instantiate the listener classes and register them in a way analogous to listeners defined at the WAR level. Unlike WAR-level listeners, the order in which the tag library listeners are registered is undefined. The only subelement of the `listener` element is the `listener-class` element, which must contain the fully qualified name of the listener class. Since the IDE does not have a listener property on the customizer, you must manually add the `listener` element to the TLD as follows:

1. Right-click the TLD.
2. Choose open
3. Uncomment the `listener` element at the bottom of the TLD.
4. Replace the dummy listener class with your listener class.

Tag Properties

Each tag in the library is described by giving its name and the class of its tag handler, information on the scripting variables created by the tag, and information on the tag's attributes.

Scripting variable information can be given directly in the TLD or through a tag extra info class (see *Tags That Define Scripting Variables*, page 121). Each attribute declaration contains an indication of whether the attribute is required, whether its value can be determined by request-time expressions, and the type of the attribute (see *Tag Attribute Properties*, page 128).

Tag properties are listed in Table 5–2:

Table 5–2 Tag Properties

Property	Description
Name	The unique tag name
Tag Class	The fully-qualified name of the tag handler class
Display Name	Optional name intended to be displayed by tools
Small Icon	Optional small-icon that can be used by tools
Large Icon	Optional large-icon that can be used by tools
Description	Optional tag-specific information

To add a tag to a tag library using the IDE:

1. Right-click the TLD file and choose Add Tag.
2. Type the tag name and tag class name.
3. Click OK.

The following sections describe the tag handler methods you need to develop and the properties you need to specify for each type of tag introduced in *Types of Tags* (page 119).

Tag Handlers

A *tag handler* is an object invoked by a Web container to evaluate a custom tag during the execution of the JSP page that references the tag. Tag handlers must implement either the `Tag` or `BodyTag` interface. Interfaces can be used to take an existing Java object and make it a tag handler. For newly created handlers, you can use the `TagSupport` and `BodyTagSupport` classes as base classes. These classes and interfaces are contained in the `javax.servlet.jsp.tagext` package.

Tag handler methods defined by the `Tag` and `BodyTag` interfaces are called by the JSP page's servlet at various points during the evaluation of the tag. When the start tag of a custom tag is encountered, the JSP page's servlet calls methods to initialize the appropriate handler and then invokes the handler's `doStartTag` method. When the end tag of a custom tag is encountered, the handler's `doEndTag` method is invoked. Additional methods are invoked in between when a tag handler needs to interact with the body of the tag. For further information, see *Tags with Bodies* (page 130). In order to provide a tag handler implementation, you must implement the methods, summarized in Table 5–3, that are invoked at various stages of processing the tag.

Table 5–3 Tag Handler Methods

Tag Handler Type	Methods
Simple	<code>doStartTag</code> , <code>doEndTag</code> , <code>release</code>
Attributes	<code>doStartTag</code> , <code>doEndTag</code> , <code>set/getAttribute1...N</code> , <code>release</code>
Body, Evaluation and No Interaction	<code>doStartTag</code> , <code>doEndTag</code> , <code>release</code>
Body, Iterative Evaluation	<code>doStartTag</code> , <code>doAfterBody</code> , <code>doEndTag</code> , <code>release</code>
Body, Interaction	<code>doStartTag</code> , <code>doEndTag</code> , <code>release</code> , <code>doInitBody</code> , <code>doAfterBody</code> , <code>release</code>

A tag handler has access to an API that allows it to communicate with the JSP page. The entry point to the API is the page context object (`javax.servlet.jsp.PageContext`), through which a tag handler can retrieve all the other implicit objects (request, session, and application) accessible from a JSP page.

Implicit objects can have named attributes associated with them. Such attributes are accessed using `[set|get]Attribute` methods.

If the tag is nested, a tag handler also has access to the handler (called the *parent*) associated with the enclosing tag.

A set of related tag handler classes (a tag library) is usually packaged and deployed as a JAR archive.

You can generate a tag handler skeleton with the IDE as follows:

1. Right-click the tag.
2. Choose Customize.
3. Select the Code Generation tab.
4. Choose whether to implement an interface or extend a class which interface or class.
5. Click OK.
6. Right-click the tag and choose Generate Tag Handlers.

Once you have the skeleton, you must implement the tag handler methods as described in the following sections.

Simple Tags

Tag Handlers

The handler for a simple tag must implement the `doStartTag` and `doEndTag` methods of the `Tag` interface. The `doStartTag` method is invoked when the start tag is encountered. This method returns `SKIP_BODY` because a simple tag has no body. The `doEndTag` method is invoked when the end tag is encountered. The `doEndTag` method needs to return `EVAL_PAGE` if the rest of the page needs to be evaluated; otherwise, it should return `SKIP_PAGE`.

The simple tag discussed in the first section,

```
<tt:simple />
```

would be implemented by the following tag handler:

```
public SimpleTag extends TagSupport {
    public int doStartTag() throws JspException {
        try {
            pageContext.getOut().print("Hello.");
        } catch (Exception ex) {
            throw new JspTagException("SimpleTag: " +
                ex.getMessage());
        }
        return SKIP_BODY;
    }
    public int doEndTag() {
        return EVAL_PAGE;
    }
}
```

Tag Properties

To indicate that a tag has an empty body:

1. Right-click the tag.
2. Choose Customize.
3. Choose empty from the Body Content drop-down list.
4. Regenerate the tag handler.

Tags with Attributes

Defining Attributes in a Tag Handler

For each tag attribute, you must define a property and get and set methods that conform to the JavaBeans architecture conventions in the tag handler. For example, the tag handler for the Struts `logic:present` tag,

```
<logic:present parameter="Clear">
```

contains the following declaration and methods:

```
protected String parameter = null;
public String getParameter() {
    return (this.parameter);
}
public void setParameter(String parameter) {
    this.parameter = parameter;
}
```

A tag attribute whose value is a `String` can name an attribute of one of the implicit objects available to tag handlers. An implicit object attribute would be accessed by passing the tag attribute value to the `[set|get]Attribute` method of the implicit object. This is a good way to pass scripting variable names to a tag handler where they are associated with objects stored in the page context (See Implicit Objects, page 92).

Tag Attribute Properties

For each tag attribute, you must specify whether the attribute is required, whether the value can be determined by an expression, and, optionally, the type of the attribute. For static values the type is always `java.lang.String`. If the value can be determined at run-time by an expression, they you must also specify the return type expected from any expression.

If a tag attribute is not required, a tag handler should provide a default value.

For example, to view the properties of the direct attribute of the `template:parameter` tag:

1. In the IDE, mount the filesystem
`<INSTALL>/j2eetutorial/examples/web/template`.
2. Expand the `template` TLD node.
3. Expand the `parameter` tag node.
4. Expand the `Attributes` node.
5. Right-click the `direct` attribute and choose `Customize`.

Notice how attribute is required, that its value can be set by a runtime expression, and that its type is `boolean`.

If you change a tag attribute, you must regenerate the tag handler.

Attribute Validation

The documentation for a tag library should describe valid values for tag attributes. When a JSP page is translated, a Web container will enforce any constraints contained in the TLD element for each attribute.

The attributes passed to a tag can also be validated at translation time with the `isValid` method of a class derived from `TagExtraInfo`. This class is also used to provide information about scripting variables defined by the tag (see *Providing Information about the Scripting Variable*, page 133).

The `isValid` method is passed the attribute information in a `TagData` object, which contains attribute-value tuples for each of the tag's attributes. Since the validation occurs at translation time, the value of an attribute that is computed at request time will be set to `TagData.REQUEST_TIME_VALUE`.

Assume that the attribute `attr1` of a tag is of type boolean and can be determined at runtime. The following `isValid` method checks that the value of `attr1` is a valid Boolean value. Note that since the value of `attr1` can be computed at runtime, `isValid` must check whether the tag user has chosen to provide a runtime value.

```
public class TwaTEI extends TagExtraInfo {
    public boolean isValid(Tagdata data) {
        Object o = data.getAttribute("attr1");
        if (o != null && o != TagData.REQUEST_TIME_VALUE) {
            if (((String)o).toLowerCase().equals("true") ||
                ((String)o).toLowerCase().equals("false") )
                return true;
            else
                return false;
        }
        else
            return true;
    }
}
```

Tags with Bodies

Tag Handlers

A tag handler for a tag with a body is implemented differently depending on whether the tag handler needs to interact with the body or not. By *interact*, we mean that the tag handler reads or modifies the contents of the body.

Tag Handler Does Not Interact with the Body

If the tag handler does not need to interact with the body, the tag handler should implement the `Tag` interface (or be derived from `TagSupport`). If the body of the tag needs to be evaluated, the `doStartTag` method needs to return `EVAL_BODY_INCLUDE`; otherwise it should return `SKIP_BODY`.

If a tag handler needs to iteratively evaluate the body, it should implement the `IterationTag` interface or be derived from `TagSupport`. It should return `EVAL_BODY_AGAIN` from the `doStartTag` and `doAfterBody` methods if it determines that the body needs to be evaluated again.

Tag Handler Interacts with the Body

If the tag handler needs to interact with the body, the tag handler must implement `BodyTag` (or be derived from `BodyTagSupport`). Such handlers typically implement the `doInitBody` and the `doAfterBody` methods. These methods interact with body content passed to the tag handler by the JSP page's servlet.

Body content supports several methods to read and write its contents. A tag handler can use the body content's `getString` or `getReader` methods to extract information from the body, and the `writeOut(out)` method to write the body contents to an out stream. The writer supplied to the `writeOut` method is obtained using the tag handler's `getPreviousOut` method. This method is used to ensure that a tag handler's results are available to an enclosing tag handler.

If the body of the tag needs to be evaluated, the `doStartTag` method needs to return `EVAL_BODY_BUFFERED`; otherwise, it should return `SKIP_BODY`.

`doInitBody` Method

The `doInitBody` method is called after the body content is set but before it is evaluated. You generally use this method to perform any initialization that depends on the body content.

`doAfterBody` Method

The `doAfterBody` method is called *after* the body content is evaluated.

Like the `doStartTag` method, `doAfterBody` must return an indication of whether to continue evaluating the body. Thus, if the body should be evaluated again, as would be the case if you were implementing an iteration tag, `doAfterBody` should return `EVAL_BODY_BUFFERED`; otherwise, `doAfterBody` should return `SKIP_BODY`.

release **Method**

A tag handler should reset its state and release any private resources in the `release` method.

The following example reads the content of the body (which contains a SQL query) and passes it to an object that executes the query. Since the body does not need to be reevaluated, `doAfterBody` returns `SKIP_BODY`.

```
public class QueryTag extends BodyTagSupport {
    public int doAfterBody() throws JspTagException {
        BodyContent bc = getBodyContent();
        // get the bc as string
        String query = bc.getString();
        // clean up
        bc.clearBody();
        try {
            Statement stmt = connection.createStatement();
            result = stmt.executeQuery(query);
        } catch (SQLException e) {
            throw new JspTagException("QueryTag: " +
                e.getMessage());
        }
        return SKIP_BODY;
    }
}
```

Tag Properties

Body content containing custom and core tags, scripting elements, and HTML text is categorized as JSP. This is the value declared for the Struts `logic:present` tag. All other types of body content—for example—SQL statements passed to the query tag, would be labeled `tagdependent`.

To indicate that a tag has a body:

1. Right-click the tag.
2. Choose `Customize`.

3. Choose the content type from the Body Content drop-down list.
4. Regenerate the tag handler.

Tags That Define Scripting Variables

Tag Handlers

A tag handler is responsible for creating and setting the object referred to by the scripting variable into a context accessible from the page. It does this by using the `pageContext.setAttribute(name, value, scope)` or `pageContext.setAttribute(name, value)` methods. Typically an attribute passed to the custom tag specifies the name of the scripting variable object; this name can be retrieved by invoking the attribute's `get` method described in Using Scope Objects (page 55).

If the value of the scripting variable is dependent on an object present in the tag handler's context, it can retrieve the object using the `pageContext.getAttribute(name, scope)` method.

The usual procedure is that the tag handler retrieves a scripting variable, performs some processing on the object, and then sets the scripting variable's value using the `pageContext.setAttribute(name, object)` method.

The scope that an object can have is summarized in Table 5–4. The scope constrains the accessibility and lifetime of the object.

Table 5–4 Scope of Objects

Name	Accessible From	Lifetime
page	Current page	Until the response has been sent back to the user or the request is passed to a new page
request	Current page and any included or forwarded pages	Until the response has been sent back to the user
session	Current request and any subsequent request from the same browser (subject to session lifetime)	The life of the user's session

Table 5–4 Scope of Objects (Continued)

Name	Accessible From	Lifetime
application	Current and any future request from the same Web application	The life of the application

Providing Information about the Scripting Variable

The last example described in *Tags That Define Scripting Variables* (page 121) defines a scripting variable name that is used for accessing information about the elements in a collection:

```
<tl:iterator name="member" type="org.Member"
  group="<%= dept.getMembers()%>">
  <tr>
    <td><jsp:getProperty name="member"
      property="name"/></td>
    <td><jsp:getProperty name="member"
      property="phone"/></td>
    <td><jsp:getProperty name="member"
      property="email"/></td>
  </tr>
</tl:iterator>
```

When the JSP page containing this tag is translated, the Web container generates code to synchronize the scripting variable with the object referenced by the variable. To generate the code, the Web container requires certain information about the scripting variable:

- Variable name
- Variable class
- Whether the variable refers to a new or existing object
- The availability of the variable.

There are two ways to provide this information: by specifying scripting variable properties in the TLD or by defining a tag extra info class and including the `tei-class` element in the TLD. Using the `variable` element is simpler, but slightly less flexible.

Scripting Variable Properties

A scripting variable has a name. The name can be specified either as a constant or via the name of an attribute whose translation-time value specifies the name of the variable.

The following scripting variable properties are optional:

- The fully qualified constant name of the class of the variable. `java.lang.String` is the default.
- Whether the variable refers to a new object. `True` is the default.
- The scope of the scripting variable defined. `NESTED` is the default. Table 5–5 describes the availability of the scripting variable and the methods where the value of the variable must be set or reset.

Table 5–5 Scripting Variable Availability

Value	Availability	Methods
NESTED	Between the start tag and the end tag	In <code>doInitBody</code> and <code>doAfterBody</code> for a tag handler implementing <code>BodyTag</code> ; otherwise, in <code>doStartTag</code>
AT_BEGIN	From the start tag until the end of the page	In <code>doInitBody</code> , <code>doAfterBody</code> , and <code>doEndTag</code> for a tag handler implementing <code>BodyTag</code> ; otherwise, in <code>doStartTag</code> and <code>doEndTag</code>
AT_END	After the end tag until the end of the page	In <code>doEndTag</code>

TagExtraInfo Class

You can also define a tag extra info class by extending the class `javax.servlet.jsp.TagExtraInfo`. A `TagExtraInfo` must implement the `getVariableInfo` method to return an array of `VariableInfo` objects containing the following information:

- Variable name
- Variable class
- Whether the variable refers to a new object
- The availability of the variable

The Web container passes a parameter called `data` to the `getVariableInfo` method that contains attribute-value tuples for each of the tag's attributes. These attributes can be used to provide the `VariableInfo` object with a scripting variable's name and class.

The iterator tag library provides information about the scripting variable created by the iterator tag in the `IteratorTEI` tag extra info class because the scripting variable customizer does not allow a tag attribute to specify the type of the variable. Since the name (`member`) and class (`org.Member`) of the scripting variable are passed in as tag attributes, they are retrieved with the `data.getAttributeString` method and used to fill in the `VariableInfo` constructor. To allow the scripting variable name to be used within the scope of the tag, the scope of book is set to be `NESTED`.

```
public class IteratorTEI extends TagExtraInfo {
    ...
    public VariableInfo[] getVariableInfo(TagData data) {
        VariableInfo info1 = new VariableInfo(
            data.getAttributeString("name"),
            data.getAttributeString("type"),
            true,
            VariableInfo.NESTED);
        VariableInfo [] info = { info1 };
        return info;
    }
}
```

Since the tag library customizer does not have a TEI class property, the fully qualified name of the tag extra info class defined for a scripting variable must be manually added to the TLD in the `tei-class` subelement of the tag element. The `tei-class` element for `IteratorTEI` would be as follows:

```
<tei-class>iterator.IteratorTEI</tei-class>
```

Cooperating Tags

Tags cooperate by sharing objects. JSP technology supports two styles of object sharing.

The first style requires that a shared object be named and stored in the page context (one of the implicit objects accessible to both JSP pages and tag handlers). To access objects created and named by another tag, a tag handler uses the `pageContext.getAttribute(name, scope)` method.

In the second style of object sharing, an object created by the enclosing tag handler of a group of nested tags is available to all inner tag handlers. This form of object sharing has the advantage that it uses a private namespace for the objects, thus reducing the potential for naming conflicts.

To access an object created by an enclosing tag, a tag handler must first obtain its enclosing tag with the static method `TagSupport.findAncestorWithClass(from, class)` or the `TagSupport.getParent` method. The former method should be used when a specific nesting of tag handlers cannot be guaranteed. Once the ancestor has been retrieved, a tag handler can access any statically or dynamically created objects. Statically created objects are members of the parent. Private objects can also be created dynamically. Such objects can be stored in a tag handler with the `setValue` method and retrieved with the `getValue` method.

The following example illustrates a tag handler that supports both the named and private object approaches to sharing objects. In the example, the handler for a query tag checks whether an attribute named `connection` has been set in the `doStartTag` method. If the `connection` attribute has been set, the handler retrieves the connection object from the page context. Otherwise, the tag handler first retrieves the tag handler for the enclosing tag, and then retrieves the connection object from that handler.

```
public class QueryTag extends BodyTagSupport {
    private String connectionId;
    public int doStartTag() throws JspException {
        String cid = getConnection();
        if (cid != null) {
            // there is a connection id, use it
            connection = (Connection)pageContext.
                getAttribute(cid);
        } else {
            ConnectionTag ancestorTag =
                (ConnectionTag)findAncestorWithClass(this,
                    ConnectionTag.class);
            if (ancestorTag == null) {
                throw new JspTagException("A query without
                    a connection attribute must be nested
                    within a connection tag.");
            }
            connection = ancestorTag.getConnection();
        }
    }
}
```

The query tag implemented by this tag handler could be used in either of the following ways:

```
<tt:connection id="con01" ....> ... </tt:connection>
<tt:query id="balances" connection="con01">
  SELECT account, balance FROM acct_table
    where customer_number = <%= request.getCustno()%>
</tt:query>

<tt:connection ....>
  <x:query id="balances">
    SELECT account, balance FROM acct_table
      where customer_number = <%= request.getCustno()%>
  </x:query>
</tt:connection>
```

The TLD for the tag handler must indicate that the connection attribute is optional with the following declaration:

```
<tag>
...
  <attribute>
    <name>connection</name>
    <required>false</required>
  </attribute>
</tag>
```

Examples

The custom tags described in this section demonstrate solutions to two recurring problems in developing JSP applications: minimizing the amount of Java programming in JSP pages and ensuring a common look and feel across applications. In doing so, they illustrate many of the styles of tags discussed in the first part of the chapter.

An Iteration Tag

Constructing page content that is dependent on dynamically generated data often requires the use of flow control scripting statements. By moving the flow control logic to tag handlers, flow control tags reduce the amount of scripting needed in JSP pages.

An iterator tag retrieves objects from a collection stored in a JavaBeans component and assigns them to a scripting variable. The body of the tag retrieves information from the scripting variable. While elements remain in the collection, the iterator tag causes the body to be reevaluated. To run the example described in this section:

1. In the IDE, mount the filesystem
`<INSTALL>/j2eetutorial/examples/web/iterator.`
2. Expand the iterator node.
3. Right-click the WEB-INF directory and choose Deploy.
4. Right-click the WEB-INF directory and choose Execute.

Your browser should display the page shown in Figure 5–2.

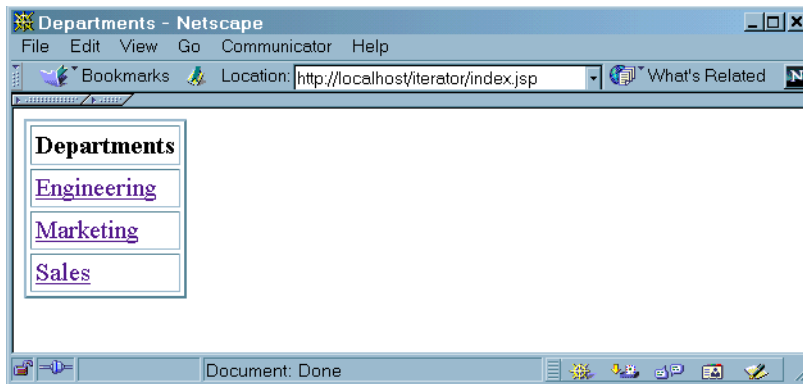


Figure 5–2 List of Departments

JSP Page

The `list.jsp` page of the iterator example uses the iterator tag to step through a collection of department members. First, the collection is retrieved by a name specified as a request parameter. The collection is supplied to the iterator tag which sets the scripting variable named `member` to each item in the collection. Finally, properties of the member variable are retrieved with `jsp:getProperty` statements for display in a table.

```
<jsp:useBean id="org" class="org.Organization"/>
<% String deptName = (String)request.getParameter("deptName");
   org.Department dept = org.getDepartment(deptName); %>
<head>
<title><%= deptName%> Department</title>
```

```

</head>
<body bgcolor="white">
<table border=2 cellspacing=3 cellpadding=3>
  <tr>
    <td colspan=3><b><center>
      <%= deptName%></b></center></td>
  </tr>
  <tr>
    <td width=100><b>Name</b></td>
    <td width=100><b>Extension</b></td>
    <td width=100><b>Email</b></td>
  </tr>
  <!-- List all department members -->
  <tl:iterator name="member" type="org.Member"
    group="<%= dept.getMembers()%>">
    <tr>
      <td><jsp:getProperty name="member"
        property="name" /></td>
      <td><jsp:getProperty name="member"
        property="phone" /></td>
      <td><jsp:getProperty name="member"
        property="email" /></td>
    </tr>
  </tl:iterator>
</table>
</body>

```

Upon selecting the Engineering department from the previous page, your browser will display the page in Figure 5–3.

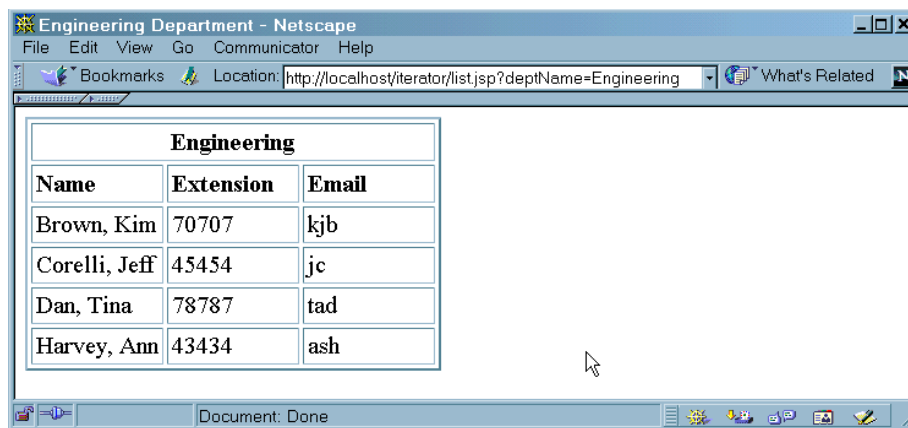


Figure 5–3 Engineering Department Contact Information

Tag Handler

The `doStartTag` method first initializes an iterator from a collection provided as a tag attribute. If the iterator contains more elements, `doStartTag` sets the value of the scripting variable to the next element and then sets the return value to `EVAL_BODY_INCLUDE` to indicate that the body should be evaluated; otherwise it ends the iteration by returning `SKIP_BODY`.

After the body has been evaluated, the `doAfterBody` method sets the value of the scripting variable to the next element and returns `EVAL_BODY_AGAIN` to indicate that the body should be evaluated again. This causes the reexecution of `doAfterBody`. When there are no remaining elements, `doAfterBody` terminates the process by returning `SKIP_BODY`.

```
public int doStartTag() throws JspException, JspException {
    otherDoStartTagOperations();

    if (theBodyShouldBeEvaluated()) {
        return EVAL_BODY_INCLUDE;
    } else {
        return SKIP_BODY;
    }
}

public void otherDoStartTagOperations()
    if(group.size() > 0)
        iterator = group.iterator();
}

public boolean theBodyShouldBeEvaluated() {
    if (iterator.hasNext()) {
        pageContext.setAttribute(name, iterator.next());
        return true;
    }
    else
        return false;
}

public int doAfterBody() throws JspException {
    if (theBodyShouldBeEvaluatedAgain()) {
        return EVAL_BODY_AGAIN;
    } else {
        return SKIP_BODY;
    }
}
```



```
public boolean theBodyShouldBeEvaluatedAgain() {
    if (iterator.hasNext()) {
        pageContext.setAttribute(name, iterator.next());
        return true;
    }
    else
        return false;
}
```

Tag Extra Info Class

Information about the scripting variable is provided in the `IteratorTEI` tag extra info class. The name and class of the scripting variable are passed in as the name and type tag attributes and used to fill in the `VariableInfo` constructor.

```
public VariableInfo[] getVariableInfo(TagData data) {
    VariableInfo info1 = new VariableInfo(
        data.getAttributeString("name"),
        data.getAttributeString("type"),
        true,
        VariableInfo.NESTED);
    VariableInfo [] info = { info1 };
    return info;
}
```

A Template Tag Library

A template provides a way to separate the common elements that are part of each screen from the elements that change with each screen of an application. Putting all the common elements together into one file makes it easier to maintain and enforce a consistent look and feel in all the screens. It also makes development of individual screens easier because the designer can focus on portions of a screen that are specific to that screen while the template takes care of the common portions.

The template is a JSP page with placeholders for the parts that need to change with each screen. Each of these placeholders is referred to as a *parameter* of the template. For example, a simple template could include a title parameter for the top of the generated screen and a body parameter to refer to a JSP page for the custom content of the screen.

The template uses a set of nested tags—definition, screen, and parameter—to define a table of screen definitions and uses an insert tag to insert parameters from a screen definition into a specific application screen.

JSP Page

The template for the Duke's Bookstore example, `template.jsp`, is shown below. This page includes a JSP page that creates the screen definition and then uses the insert tag to insert parameters from the definition into the application screen.

```
<%@ taglib uri="/template" prefix="tt" %>
<%@ page errorPage="errorpage.jsp" %>
<%@ include file="screendefinitions.jsp" %><html>
  <head>
    <title>
      <tt:insert definition="bookstore"
        parameter="title"/>
    </title>
  </head>
  <tt:insert definition="bookstore"
    parameter="banner"/>
  <tt:insert definition="bookstore"
    parameter="body"/>
</body>
</html>
```

`screendefinitions.jsp` creates a screen definition based on a request attribute `selectedScreen`:

```
<tt:definition name="bookstore"
  screen="<%= (String)request.
    getAttribute(\"selectedScreen\") %>">
  <tt:screen id="/enter">
    <tt:parameter name="title"
      value="Duke's Bookstore" direct="true"/>
    <tt:parameter name="banner"
      value="/banner.jsp" direct="false"/>
    <tt:parameter name="body"
      value="/bookstore.jsp" direct="false"/>
  </tt:screen>
  <tt:screen id="/catalog">
    <tt:parameter name="title"
```

```

        value="<%=messages.getString("TitleBookCatalog")%>"
        direct="true"/>
        ...
    </tt:definition>

```

The template is instantiated by the `Dispatcher` servlet. `Dispatcher` first gets the requested screen and stores it as an attribute of the request. This is necessary because when the request is forwarded to `template.jsp`, the request URL doesn't contain the original request (for example, `/bookstore3/catalog`) but instead reflects the path (`/bookstore3/template.jsp`) of the forwarded page. Finally, the servlet dispatches the request to `template.jsp`:

```

public class Dispatcher extends HttpServlet {
    public void doGet(HttpServletRequest request,
        HttpServletResponse response) {
        request.setAttribute("selectedScreen",
            request.getServletPath());
        RequestDispatcher dispatcher =
            request.getRequestDispatcher("/template.jsp");
        if (dispatcher != null)
            dispatcher.forward(request, response);
    }
    public void doPost(HttpServletRequest request,
        HttpServletResponse response) {
        request.setAttribute("selectedScreen",
            request.getServletPath());
        RequestDispatcher dispatcher =
            request.getRequestDispatcher("/template.jsp");
        if (dispatcher != null)
            dispatcher.forward(request, response);
    }
}

```

Tag Handlers

The template tag library contains four tag handlers—`DefinitionTag`, `ScreenTag`, `ParameterTag`, and `InsertTag`—that demonstrate the use of cooperating tags. `DefinitionTag`, `ScreenTag`, and `ParameterTag` comprise a set of nested tag handlers that share public and private objects. `DefinitionTag` creates a public object named `definition` that is used by `InsertTag`.

In `doStartTag`, `DefinitionTag` creates a public object named `screens` that contains a hash table of screen definitions. A screen definition consists of a screen identifier and a set of parameters associated with the screen.

```

public int doStartTag() {
    HashMap screens = null;
    screens = (HashMap) pageContext.getAttribute("screens",
        pageContext.APPLICATION_SCOPE);
    if (screens == null)
        pageContext.setAttribute("screens", new HashMap(),
            pageContext.APPLICATION_SCOPE);
    return EVAL_BODY_INCLUDE;
}

```

The table of screen definitions is filled in by `ScreenTag` and `ParameterTag` from text provided as attributes to these tags. Table 5–6 shows the contents of the screen definitions hash table for the Duke’s Bookstore application.

Table 5–6 Screen Definitions

Screen Id	Title	Banner	Body
/enter	Duke’s Bookstore	/banner.jsp	/bookstore.jsp
/catalog	Book Catalog	/banner.jsp	/catalog.jsp
/bookdetails	Book Description	/banner.jsp	/bookdetails.jsp
/showcart	Shopping Cart	/banner.jsp	/showcart.jsp
/cashier	Cashier	/banner.jsp	/cashier.jsp
/receipt	Receipt	/banner.jsp	/receipt.jsp

In `doEndTag`, `DefinitionTag` creates a public object of class `Definition`, selects a screen definition from the `screens` object based on the URL passed in the request, and uses it to initialize the `Definition` object.

```

public int doEndTag() throws JspTagException {
    try {
        Definition definition = new Definition();
        HashMap screens = null;
        ArrayList params = null;
        TagSupport screen = null;
        screens = (HashMap)
            pageContext.getAttribute("screens",
                pageContext.APPLICATION_SCOPE);
        if (screens != null)
            params = (ArrayList) screens.get(screenId);
    }
}

```

```

else
    ...
    if (params == null)
        ...
        Iterator ir = null;
        if (params != null)
            ir = params.iterator();
        while ((ir != null) && ir.hasNext())
            definition.setParam((Parameter) ir.next());
        // put the definition in the page context
        pageContext.setAttribute(
            definitionName, definition);
    } catch (Exception ex) {
        ex.printStackTrace();
    }
    return EVAL_PAGE;
}

```

If the URL passed in the request is `/enter`, the Definition contains the items shown in Table 5–7:

Table 5–7 Definition for the URL `/enter`

Title	Banner	Body
Duke's Bookstore	/banner.jsp	/bookstore.jsp

The screen definition for the URL `/enter` is shown in Table 5–8. The definition specifies that the value of the Title parameter, Duke's Bookstore, should be inserted directly into the output stream, but the values of Banner and Body should be dynamically included.

Table 5–8 Screen Definition for the URL `/enter`

Parameter Name	Parameter Value	isDirect
title	Duke's Bookstore	true
banner	/banner.jsp	false
body	/bookstore.jsp	false

InsertTag uses Definition to insert parameters of the screen definition into the response. In the doStartTag method, it retrieves the definition object from the page context.

```
public int doStartTag() {
    // get the definition from the page context
    definition = (Definition) pageContext.
        getAttribute(definitionName);
    // get the parameter
    if (parameterName != null && definition != null)
        parameter = (Parameter)definition.
            getParam(parameterName);
    if (parameter != null)
        directInclude = parameter.isDirect();
    return SKIP_BODY;
}
```

The doEndTag method inserts the parameter value. If the parameter is direct, it is directly inserted into the response; otherwise, the request is sent to the parameter, and the response is dynamically included into the overall response.

```
public int doEndTag() throws JspTagException {
    try {
        if (directInclude && parameter != null)
            pageContext.getOut().print(parameter.getValue());
        else {
            if ((parameter != null) &&
                (parameter.getValue() != null))
                pageContext.include(parameter.getValue());
        }
    } catch (Exception ex) {
        throw new JspTagException(ex.getMessage());
    }
    return EVAL_PAGE;
}
```

How Is a Tag Handler Invoked?

The Tag interface defines the basic protocol between a tag handler and a JSP page's servlet. It defines the life cycle and the methods to be invoked when the start and end tags are encountered.

The JSP page's servlet invokes the setPageContext, setParent, and attribute setting methods before calling doStartTag. The JSP page's servlet also guarantees that release will be invoked on the tag handler before the end of the page.

Here is a typical tag handler method invocation sequence:

```
ATag t = new ATag();
t.setPageContext(...);
t.setParent(...);
t.setAttribute1(value1);
t.setAttribute2(value2);
t.doStartTag();
t.doEndTag();
t.release();
```

The `BodyTag` interface extends `Tag` by defining additional methods that let a tag handler access its body. The interface provides three new methods:

- `setBodyContent`—Creates body content and adds to the tag handler
- `doInitBody`—Called before evaluation of the tag body
- `doAfterBody`—Called after evaluation of the tag body

A typical invocation sequence is:

```
t.doStartTag();
out = pageContext.pushBody();
t.setBodyContent(out);
// perform any initialization needed after body content is set
t.doInitBody();
t.doAfterBody();
// while doAfterBody returns EVAL_BODY_BUFFERED we
// iterate body evaluation
...
t.doAfterBody();
t.doEndTag();
t.pageContext.popBody();
t.release();
```

JavaServer Pages Standard Tag Library

Stephanie Bodoff

THE JavaServer Pages Standard Tag Library (JSTL) encapsulates core functionality common to many JSP applications. For example, instead of iterating over lists using a scriptlet or different iteration tags from numerous vendors, JSTL defines a standard tag that works the same everywhere. This standardization lets you learn a single tag and use it on multiple JSP containers. Also, when tags are standard, containers can optimize their implementation.

JSTL has support for common, structural tasks such as iteration and conditionals, tags for manipulating XML documents, internationalization tags, and tags for accessing databases using SQL. It also introduces the concept of an expression language to simplify page development. JSTL also provides a framework for integrating existing tag libraries with JSTL.

This chapter demonstrates the JSTL through excerpts from the JSP version of the Duke's Bookstore application discussed in previous chapters. It assumes that you are familiar with the material in the Using Tags (page 118) section of Chapter 5.

The Example JSP Pages

This chapter illustrates JSTL with excerpts from the JSP version of the Duke's Bookstore application discussed in Chapter 5 rewritten as follows:

- The Struts logic tags were replaced with JSTL core tags.
- The scriptlets accessing a message store were replaced with message formatting tags.
- The JavaBeans component database helper object was replaced with direct calls to the database via the JSTL SQL tags. For most applications, it is better to encapsulate calls to a database in a bean. JSTL includes SQL tags for situations where a new application is being prototyped and the overhead of creating a bean may not be warranted.

The source for the Duke's Bookstore application is located in the `<INSTALL>/j2eetutorial/examples/web/bookstore4` directory created when you unzip the tutorial bundle (see About the Examples, page ix).

To deploy and run the example:

1. In the IDE, mount the filesystem
`<INSTALL>/j2eetutorial/examples/web/bookstore4`.
2. Expand the bookstore4 node.
3. Right-click the WEB-INF directory and choose Deploy.
4. Set up the PointBase database as described in Accessing Databases from Web Applications, page 39.
5. Open the bookstore URL `http://localhost:80/bookstore4/enter`.

To review the deployment settings:

1. Expand the WEB-INF node.
2. Select the `web.xml` file.
3. Select the Deployment property sheet.
4. Browse the JSTL basename context parameter.
 - a. Click the Context Parameters property and open the property editor.
 - b. Notice that `javax.servlet.jsp.jstl.fmt.localizationContext` is set to `messages.BookstoreMessages`.
5. Browse the servlet definition and servlet mappings.
 - a. Click the Servlets property and open the property editor.

- b. Notice that the Dispatcher servlet is implemented by the Dispatcher class and that the URLs `/enter`, `/catalog`, `/bookdetails`, `/showcart`, `/cashier`, and `/receipt` are mapped to the Dispatcher servlet.
6. Browse the tag libraries mapping.
 - a. Click the Tag Libraries property and open the property editor.
 - b. Notice that the relative URI `/template` is mapped to `/WEB-INF/lib/template.jar`.
7. Browse the resource references.
 - a. Select the Resources Property sheet.
 - a. Click the Resource References property and open the property editor.
 - b. Note the resource reference named `jdbc/BookDB`.

Using JSTL

JSTL includes a wide variety of tags that naturally fit into discrete functional areas. Therefore, JSTL is exposed via multiple URIs to clearly show the functional areas it covers and give each area its own namespace. Table 6–1 summa-

rizes these functional areas, subfunctions in each area, tags in each subfunction, and the prefixes used in the Duke's Bookstore application.

Table 6–1 JSTL Tags

Area	Function	Tags	Prefix
Core	Expression Language Support	catch out remove set	c
	Flow Control	choose when otherwise forEach forEachTokens if	
	URL Management	import param redirect param url param	
XML	Core	out parse set	x
	Flow Control	choose when otherwise forEach if	
	Transformation	transform param	

Table 6–1 JSTL Tags (Continued)

Area	Function	Tags	Prefix
I18n	Locale	setLocale	fmt
	Message formatting	bundle message param setBundle	
	Number and date formatting	formatNumber formatDate parseDate parseNumber setTimeZone timeZone	
Database		setDataSource	sql
	SQL	query dateParam param transaction update dateParam param	

The URIs used to access the libraries are:

- Core: <http://java.sun.com/jstl/core>
- XML: <http://java.sun.com/jstl/xml>
- Internationalization: <http://java.sun.com/jstl/fmt>
- SQL: <http://java.sun.com/jstl/sql>

The JSTL tag libraries comes in two versions (see Twin Libraries, page 154). The URIs for the JSTL-EL library are as shown above. The URIs for the JSTL-RT library are named `append_rt` to the end.

Expression Language Support

A primary feature of JSTL is its support for an expression language (EL). An expression language, in concert with JSTL tags, makes it possible to easily access application data and manipulate it in simple ways without having to use

scriptlets or request-time expressions. Currently, a page author has to use an expression `<%= aName %>` to access the value of a system or user-defined JavaBeans component. For example:

```
<x:aTag att="<%= pageContext.getAttribute("aName") %>">
```

Referring to nested bean properties is even more complex:

```
<%= aName.getFoo().getBar() %>
```

This makes page authoring more complicated than it need be.

An expression language allows a page author to access an object using a simplified syntax such as

```
<x:atag att="${aName}">
```

for a simple variable or

```
<x:aTag att="${aName.foo.bar}">
```

for a nested property.

The JSTL expression language promotes JSP *scoped attributes* as the standard way to communicate information from business logic to JSP pages. For example, the `test` attribute of the `this` conditional tag is supplied with an expression that compares the number of items in the session-scoped attribute named `cart` with 0:

```
<c:if test="${sessionScope.cart.numberOfItems > 0}">
  ...
</c:if>
```

The next version of the JSP specification will standardize on an expression language for all custom tag libraries. This release of JSTL includes a snapshot of that expression language.

Twin Libraries

The JSTL tag libraries come in two versions which differ only in the way they support the use of runtime expressions for attribute values.

In the JSTL-RT tag library, expressions are specified in the page's scripting language. This is exactly how things currently work in current tag libraries.

In the JSTL-EL tag library, expressions are specified in the JSTL expression language. An expression is a `String` literal in the syntax of the EL.

When using the EL tag library you cannot pass a scripting language expression for the value of an attribute. This rule makes it possible to validate the syntax of an expression at translation time.

JSTL Expression Language

The JSTL expression language is responsible for handling both expressions and literals. Expressions are enclosed by the `${ }` characters. For example:

```
<c:if test="${bean1.a < 3}" />
```

Any value that does not begin with `${` is treated as a literal that is parsed to the expected type using the `PropertyEditor` for the expected type:

```
<c:if test="true" />
```

Literal values that contain the `${` characters must be escaped as follows:

```
<mytags:example attr1="an expression is ${'${'}true}" />
```

Attributes

Attributes are accessed by name, with an optional scope. Properties of attributes are accessed using the `.` operator, and may be nested arbitrarily.

The EL unifies the treatment of the `.` and `[]` operators. Thus, `expr-a.expr-b` is equivalent to `expr-a[expr-b]`. To evaluate `expr-a[expr-b]`, evaluate `expr-a` into `value-a` and evaluate `expr-b` into `value-b`.

- If `value-a` is a `Map` return `value-a.get(value-b)`.
- If `value-a` is a `List` or array coerce `value-b` to `int` and return `value-a.get(value-b)` or `Array.get(value-a, value-b)`, as appropriate.
- If `value-a` is a `JavaBeans` object, coerce `value-b` to `String`. If `value-b` is a readable property of `value-a` the return result of getter call.

The EL evaluates an identifier by looking up its value as an attribute, according to the behavior of `PageContext.findAttribute(String)`. For example, `${product}` will look for the attribute named `product`, searching the page, request, session, and application scopes and will return its value. If the attribute is not found, `null` is returned. Note that an identifier that matches one of the implicit objects described in the next section will return that implicit object instead of an attribute value.

Implicit Objects

The JSTL expression language defines a set of implicit objects:

- `pageContext` - the `PageContext` object
- `pageScope` - a `Map` that maps page-scoped attribute names to their values
- `requestScope` - a `Map` that maps request-scoped attribute names to their values
- `sessionScope` - a `Map` that maps session-scoped attribute names to their values
- `applicationScope` - a `Map` that maps application-scoped attribute names to their values
- `param` - a `Map` that maps parameter names to a single `String` parameter value (obtained by calling `ServletRequest.getParameter(String)`)
- `paramValues` - a `Map` that maps parameter names to a `String[]` of all values for that parameter (obtained by calling `ServletRequest.getParameterValues(String)`)
- `header` - a `Map` that maps header names to a single `String` header value (obtained by calling `ServletRequest.getHeader(String)`)
- `headerValues` - a `Map` that maps header names to a `String[]` of all values for that parameter (obtained by calling `ServletRequest.getHeaders(String)`)
- `cookie` - a `Map` that maps cookie names to a single `Cookie` (obtained by calling `HttpServletRequest.getCookie(String)`)
- `initParam` - a `Map` that maps a parameter names to a single `String` parameter value (obtained by calling `ServletRequest.getInitParameter(String)`)

When an expression references one of these objects by name, the appropriate object is returned instead of the corresponding attribute. For example: `${pageContext}` returns the `PageContext` object, even if there is an existing `pageCon-`

text attribute containing some other value. Table 6–2 shows some examples of using these implicit objects.

Table 6–2 Example JSTL Expressions

Expression	Result
<code>\${pageContext.request.contextPath}</code>	The context path (obtained from <code>HttpServletRequest</code>)
<code>\${sessionScope.cart.numberOfItems}</code>	The <code>numberOfItems</code> property of the session-scoped attribute named <code>cart</code>
<code>\${param["mycom.productId"]}</code>	The <code>String</code> value of the <code>mycom.productId</code> parameter

Literals

- Boolean: `true` and `false`
- Long: as in Java
- Floating point: as in Java
- String: with single and double quotes. `"` is escaped as `\`, `'` is escaped as `\'`, and `\` is escaped as `\\`.
- Null: `null`

Operators

The EL provides the following operators:

- Arithmetic: `+`, `-`, `*`, `/` and `div`, `%` and `mod`, `-`
- Logical: `and`, `&&`, `or`, `||`, `not`, `!`
- Relational: `==`, `eq`, `!=`, `ne`, `<`, `lt`, `>`, `gt`, `<=`, `ge`, `>=`, `le`. Comparisons may be made against other values, or against boolean, string, integer, or floating point literals.
- Empty: The empty operator is a prefix operation that can be used to determine if a value is `null` or empty.

Consult the JSTL 1.0 Specification for the precedence and effects of these operators.

Tag Collaboration

Tags usually collaborate with their environment in implicit and explicit ways. Implicit collaboration is done via a well defined interface that allows nested tags to work seamlessly with the ancestor tag exposing that interface. The JSTL iterator tags support this mode of collaboration.

Explicit collaboration happens when a tag exposes information to its environment. Traditionally, this has been done by exposing a scripting variable (with a JSP scoped attribute providing the actual object). Because JSTL has an expression language, there is less need for scripting variables. So the JSTL tags (both the EL and RT versions) expose information only as JSP scoped attributes; no scripting variables are used. The convention JSTL follows is to use the name `var` for any tag attribute that exports information about the tag. For example, the `forEach` tag exposes the current item of the shopping cart it is iterating over in the following way:

```
<c:forEach var="item" items="${sessionScope.cart.items}">
  ...
</c:forEach>
```

The name `var` was selected to highlight the fact that the scoped variable exposed is not a scripting variable (which is normally the case for attributes named `id`).

In situations where a tag exposes more than one piece of information, the name `var` is used for the primary piece of information being exported, and an appropriate name is selected for any other secondary piece of information exposed. For example, iteration status information is exported by the `forEach` tag via the attribute `status`.

Core Tags

The core tags include those related to expressions, flow control, and a generic way to access URL-based resources whose content can then be included or processed within the JSP page.

Table 6–3 Core Tags

Area	Function	Tags
Core	Expression Language Support	catch out remove set
	Flow Control	choose when otherwise forEach forTokens if
	URL Management	import param redirect param url param

Expression Tags

The `out` tag evaluates an expression and outputs the result of the evaluation to the current `JspWriter` object. It is the equivalent of the JSP syntax `<%= expression %>`. For example, `showcart.jsp` displays the number of items in a shopping cart as follows:

```
<c:out value="${sessionScope.cart.numberOfItems}"/>
```

The `set` tag sets the value of an attribute in any of the JSP scopes (page, request, session, application). If the attribute does not already exist, it is created.

The JSP scoped attribute can be set either from attribute value:

```
<c:set var="foo" scope="session" value="..." />
```

or from the body of the tag:

```
<c:set var="foo">
    ...
</c:set>
```

For example, the following sets a page-scoped attribute named bookID with the value of the request parameter named Remove:

```
<c:set var="bookId" value="${param.Remove}" />
```

If you were using the RT version of the library, the statement would be:

```
<c_rt:set var="bookId"
    value="<%= request.getParameter("Remove") %>" />
```

To remove a scoped attribute, you use the remove tag. When the bookstore JSP page receipt.jsp is invoked, the shopping session is finished, so the cart session attribute is removed as follows:

```
<c:remove var="cart" scope="session" />
```

The JSTL expression language reduces the need for scripting. However, page authors will still have to deal with situations where some attributes of non-JSTL tags must be specified as expressions in the page's scripting language. The standard JSP element `jsp:useBean` is used to declare a scripting variable that can be used in a scripting language expression or scriptlet. For example, `showcart.jsp` removes a book from a shopping cart using a scriptlet. The ID of the book to be removed is passed as a request parameter. The value of the request parameter is first set as a page attribute (to be used later by the JSTL `sql:query` tag) and then declared as scripting variable and passed to the `cart.remove` method:

```
<c:set var="bookId" value="${param.Remove}" />
<jsp:useBean id="bookId" type="java.lang.String" />
<% cart.remove(bookId); %>
<sql:query var="books"
    dataSource="${applicationScope.bookDS}"
    select * from PUBLIC.books where id = ?
    <sql:param value="${bookId}" />
</sql:query>
```

The `catch` tag provides a complement to the JSP error page mechanism. It allows page authors to recover gracefully from error conditions that they can control. Actions that are of central importance to a page should *not* be encapsulated in a `catch`, so their exceptions will propagate to an error page. Actions with secondary importance to the page should be wrapped in a `catch`, so they never cause the error page mechanism to be invoked.

The exception thrown is stored in the scoped variable identified by `var`, which always has page scope. If no exception occurred, the scoped variable identified by `var` is removed if it existed. If `var` is missing, the exception is simply caught and not saved.

Flow Control Tags

To execute flow control logic, a page author must generally resort to using scriptlets. For example, the following scriptlet is used to iterate through a shopping cart:

```
<%
    Iterator i = cart.getItems().iterator();
    while (i.hasNext()) {
        ShoppingCartItem item =
            (ShoppingCartItem)i.next();
        ...
    }
%>
|  |
| --- |
| <%=item.getQuantity()%> |

...

```

Flow control tags eliminate the need for scriptlets. The next two sections have examples that demonstrate the conditional and iterator tags.

Conditional Tags

The `if` tag allows the conditional execution of its body according to value of a test attribute. The following example from `catalog.jsp` tests whether the request parameter `Add` is empty. If the test evaluates to `true`, the page queries the

database for the book record identified by the request parameter and adds the book to the shopping cart:

```
<c:if test="${!empty param.Add}">
  <c:set var="bid" value="${param.Add}"/>
  <jsp:useBean id="bid" type="java.lang.String" />
  <sql:query var="books"
    dataSource="${applicationScope.bookDS}"
    select * from PUBLIC.books where id = ?
  <sql:param value="${bid}" />
</sql:query>
<c:forEach var="bookRow" begin="0" items="${books.rows}">
  <jsp:useBean id="bookRow" type="java.util.Map" />
  <jsp:useBean id="addedBook"
    class="database.BookDetails" scope="page" />
  ...
  <% cart.add(bid, addedBook); %>
  ...
</c:if>
```

The `choose` tag performs conditional block execution by the embedded when sub tags. It renders the body of the first when tag whose test condition evaluates to true. If none of the test conditions of nested when tags evaluate to true, then the body of an otherwise tag is evaluated, if present.

For example, the following sample code shows how to render text based on a customer's membership category.

```
<c:choose>
  <c:when test="${customer.category == 'trial'}" >
    ...
  </c:when>
  <c:when test="${customer.category == 'member'}" >
    ...
  </c:when>
  <c:when test="${customer.category == 'preferred'}" >
    ...
  </c:when>
  <c:otherwise>
    ...
  </c:otherwise>
</c:choose>
```

The `choose`, `when`, and `otherwise` tags can be used to construct an if-then-else statement as follows:

```
<c:choose>
  <c:when test="${count == 0}" >
    No records matched your selection.
  </c:when>
  <c:otherwise>
    <c:out value="${count}"/> records matched your selection.
  </c:otherwise>
</c:choose>
```

Iterator Tags

The `forEach` tag allows you to iterate over a collection of objects. You specify the collection via the `items` attribute, and the current item is available through a scope variable named by the `item` attribute.

A large number of collection types are supported by `forEach`, including all implementations of `java.util.Collection` and `java.util.Map`. If the `items` attribute is of type `java.util.Map`, then the current item will be of type `java.util.Map.Entry`, which has the following properties:

- `key` - the key under which the item is stored in the underlying `Map`
- `value` - the value that corresponds to the key

Arrays of objects as well as arrays of primitive types (for example, `int`) are also supported. For arrays of primitive types, the current item for the iteration is automatically wrapped with its standard wrapper class (for example, `Integer` for `int`, `Float` for `float`, and so on).

Implementations of `java.util.Iterator` and `java.util.Enumeration` are supported but these must be used with caution. `Iterator` and `Enumeration` objects are not resettable so they should not be used within more than one iteration tag. Finally, `java.lang.String` objects can be iterated over if the string contains a list of comma separated values (for example: `Monday,Tuesday,Wednesday,Thursday,Friday`).

Here's the shopping cart iteration from the previous section with the `forEach` tag:

```
<c:forEach var="item" items="${sessionScope.cart.items}">
  ...
  <tr>
    <td align="right" bgcolor="#ffffff">
```

```

        <c:out value="${item.quantity}"/>
    </td>
    ...
</c:forEach>

```

The `forTokens` tag is used to iterate over a collection of tokens separated by a delimiter.

URL Tags

The `jsp:include` element provides for the inclusion of static and dynamic resources in the same context as the current page. However, `jsp:include` cannot access resources that reside outside of the Web application and causes unnecessary buffering when the resource included is used by another element.

In the example below, the `transform` element uses the content of the included resource as the input of its transformation. The `jsp:include` element reads the content of the response, writes it to the body content of the enclosing `transform` element, which then re-reads the exact same content. It would be more efficient if the `transform` element could access the input source directly and avoid the buffering involved in the body content of the `transform` tag.

```

<acme:transform>
    <jsp:include page="/exec/employeesList"/>
</acme:transform>

```

The `import` tag is therefore the simple, generic way to access URL-based resources whose content can then be included and or processed within the JSP page. For example, in XML Tags (page 165), `import` is used to read in the XML document containing book information and assign the content to the scoped variable `xml`:

```

<c:import url="/books.xml" var="xml" />
<x:parse xml="${xml}" var="booklist"
    scope="application" />

```

The `param` tag, analogous to the `jsp:param` tag (see `jsp:param` Element, page 100), can be used with `import` to specify request parameters.

In Session Tracking (page 78) we discussed how an application must rewrite URLs to enable session tracking whenever the client turns off cookies. You can use the `url` tag to rewrite URLs returned from a JSP page. The tag includes the session ID in the URL only if cookies are disabled; otherwise, it returns the URL

unchanged. Note that this feature requires the URL to be *relative*. The `url` tag takes `param` subtags for including parameters in the returned URL. For example, `catalog.jsp` rewrites the URL used to add a book to the shopping cart as follows:

```
<c:url var="url"
  value="/catalog" >
  <c:param name="Add" value="${bookId}" />
</c:url>
<p><strong><a href="<c:out value='${url}' />">
```

The `redirect` tag sends an HTTP redirect to the client. The `redirect` tag takes `param` subtags for including parameters in the returned URL.

XML Tags

A key aspect of dealing with XML documents is to be able to easily access their content. XPath, a W3C recommendation since 1999, provides an easy notation for specifying and selecting parts of an XML document. The JSTL XML tag set, listed in Table 6–4, is based on XPath.

Table 6–4 XML Tags

Area	Function	Tags
XML	Core	out parse set
	Flow Control	choose when otherwise forEach if
	Transformation	transform param

The XML tags use XPath as a *local* expression language; XPath expressions are always specified using attribute `select`. This means that only values specified for `select` attributes are evaluated using the XPath expression language. All

other attributes are evaluated using the rules associated with the global expression language.

In addition to the standard XPath syntax, the JSTL XPath engine supports the following scopes to access Web application data within an XPath expression:

- `$foo`
- `$param:`
- `$header:`
- `$cookie:`
- `$initParam:`
- `$pageScope:`
- `$requestScope:`
- `$sessionScope:`
- `$applicationScope:`

These scopes are defined in exactly the same way as their counterparts in the JSTL expression language discussed in Implicit Objects (page 156). Table 6–5 shows some examples of using the scopes.

Table 6–5 Example XPath Expressions

XPath Expression	Result
<code>\$sessionScope:profile</code>	The session-scoped attribute named <code>profile</code>
<code>\$initParam:mycom.productId</code>	The String value of the <code>mycom.productId</code> context parameter

The XML tags are illustrated in another version (`bookstore5`) of the Duke's Bookstore application. This version replaces the database with an XML representation (`books.xml`) of the bookstore database. To build and install this version of the application, follow the directions in The Example JSP Pages (page 150) replacing `bookstore4` with `bookstore5`.

Since the XML tags require an XPath evaluator, which is provided in two libraries, `jaxen-full.jar` and `saxpath.jar`, included with the JSTL tag library.

Core Tags

The core XML tags provide basic functionality to easily parse and access XML data.

The `parse` tag parses an XML document and saves the resulting object in the scoped attribute specified by attribute `var`. In `bookstore5`, the XML document is parsed and saved to a context attribute in `parseBooks.jsp`, which is included by all JSP pages that need access to the document:

```
<c:if test="${applicationScope:booklist == null}" >
  <c:import url="/books.xml" var="xml" />
  <x:parse xml="${xml}" var="booklist" scope="application" />
</c:if>
```

The `out` and `set` tags parallel the behavior described in Expression Tags (page 159) for the XPath local expression language. The `out` tag evaluates an XPath expression on the current context node and outputs the result of the evaluation to the current `JspWriter` object.

The `set` tag evaluates an XPath expression and sets the result into a JSP scoped attribute specified by attribute `var`.

The JSP page `bookdetails.jsp` selects a book element whose `id` attribute matches the request parameter `bookId` and sets the `abook` attribute. The `out` tag then selects the book's title element and outputs the result.

```
<x:set var="abook"
select="$applicationScope.booklist/
  books/book[@id=$param:bookId]" />
<h2><x:out select="$abook/title"/></h2>
```

As you have just seen, `x:set` stores an internal XML representation of a *node* retrieved using an XPath expression; it doesn't convert the selected node into a `String` and store it. Thus, `x:set` is primarily useful for storing parts of documents for later retrieval.

If you want to store a `String`, you need to use `x:out` within `c:set`. The `x:out` tag converts the node to a `String`, and `c:set` then stores the `String` as a

scoped attribute. For example, `bookdetails.jsp` stores a scoped attribute containing a book price, which is later fed into a `fmt` tag, as follows:

```
<c:set var="price">
  <x:out select="$abook/price"/>
</c:set>
<h4><fmt:message key="ItemPrice"/>:
  <fmt:formatNumber value="{price}" type="currency"/>
```

The other option, which is more direct but requires that the user have more knowledge of XPath, is to coerce the node to a `String` manually using XPath's `string` function.

```
<x:set var="price" select="string($abook/price)"/>
```

Flow Control Tags

The XML flow control tags parallel the behavior described in Flow Control Tags (page 161) for the XPath expression language.

The JSP page `catalog.jsp` uses the `forEach` tag to display all the books contained in `booklist` as follows:

```
<x:forEach var="book"
  select="$applicationScope:booklist/books/*">
  <tr>
    <c:set var="bookId">
      <x:out select="$book/@id"/>
    </c:set>=
    <td bgcolor="#ffffaa">
      <c:url var="url"
        value="/bookdetails" >
        <c:param name="bookId" value="{bookId}" />
        <c:param name="Clear" value="0" />
      </c:url>
      <a href="{c:out value='{url}'}/">
        <strong><x:out select="$book/title"/>&nbsp;
      </strong></a></td>
    <td bgcolor="#ffffaa" rowspan=2>
      <c:set var="price">
        <x:out select="$book/price"/>
      </c:set>
      <fmt:formatNumber value="{price}" type="currency"/>
      &nbsp;
    </td>
```

```

<td bgcolor="#ffffaa" rowspan=2>
<c:url var="url" value="/catalog" >
  <c:param name="Add" value="{bookId}" />
</c:url>
<p><strong><a href="<c:out value='{url}' />">&nbsp;
  <fmt:message key="CartAdd"/>&nbsp;</a>
</td>
</tr>
<tr>
  <td bgcolor="#ffffff">
    &nbsp;&nbsp;<fmt:message key="By"/> <em>
      <x:out select="$book/firstname"/>&nbsp;
      <x:out select="$book/surname"/></em></td></tr>
</x:forEach>

```

Transformation Tags

The `transform` tag applies a transformation, specified by a XSLT stylesheet set by the attribute `xslt`, to an XML document, specified by the attribute `xml`. If the `xml` attribute is not specified, the input XML document is read from the tag's body content.

The `param` subtag can be used along with `transform` to set transformation parameters. The attributes `name` and `value` are used to specify the parameter. The `value` attribute is optional. If it is not specified the value is retrieved from the tag's body.

Internationalization Tags

In *Internationalizing and Localizing Web Applications* (page 38) we discussed the how to adapt Web applications to the language and formatting conventions of client locales. This section describes tags that support the internationalization of JSP pages.

JSTL defines tags for:

- Setting the locale for a page
- Creating locale-sensitive messages

- Formatting and parsing data elements such as numbers, currencies, dates, and times in a locale-sensitive or customized manner

Table 6–6 Internationalization Tags

Area	Function	Tags
I18n	Setting Locale	setLocale requestEncoding
	Messaging	bundle message param setBundle
	Number and Date Formatting	formatNumber formatDate parseDate parseNumber setTimeZone timeZone

Setting the Locale

The `setLocale` tag is used to override the client-specified locale for a page. The `requestEncoding` tag is used to set the request's character encoding, in order to be able to correctly decode request parameter values whose encoding is different from ISO-8859-1.

Messaging Tags

By default, browser-sensing capabilities for locales are enabled. This means that the client determines (via its browser settings) which locale to use, and allows page authors to cater to the language preferences of their clients.

bundle Tag

You use the `bundle` tag to specify a resource bundle for a page.

To define a resource bundle for a Web application you specify the context parameter `javax.servlet.jsp.jstl.fmt.localizationContext` in the Web appli-

cation deployment descriptor. Here is the declaration from the Duke's Bookstore descriptor:

```
<context-param>
  <param-name>
    javax.servlet.jsp.jstl.fmt.localizationContext
  </param-name>
  <param-value>messages.BookstoreMessages</param-value>
</context-param>
```

message Tag

The message tag is used to output localized strings. The following tag from catalog.jsp

```
<h3><fmt:message key="Choose"/></h3>
```

is used to output a string inviting customers to choose a book from the catalog.

The param subtag provides a single argument (for parametric replacement) to the compound message or pattern in its parent message tag. One param tag must be specified for each variable in the compound message or pattern. Parametric replacement takes place in the order of the param tags.

Formatting Tags

JSTL provides a set of tags for parsing and formatting locale-sensitive numbers and dates.

The formatNumber tag is used to output localized numbers. The following tag from showcart.jsp

```
<fmt:formatNumber value="${book.price}" type="currency"/>
```

is used to display a localized price for a book. Note that since the price is maintained in the database in dollars, the localization is somewhat simplistic, because the formatNumber tag is unaware of exchange rates. The tag formats currencies but does not convert them.

Analogous tags for formatting dates (formatDate), and parsing numbers and dates (parseNumber, parseDate) are also available. The timeZone tag estab-

lishes the time zone (specified via the value attribute) to be used by any nested `formatDate` tags.

In `receipt.jsp`, a “pretend” ship date is created and then formatted with the `formatDate` tag:

```
<jsp:useBean id="now" class="java.util.Date" />
<jsp:setProperty name="now" property="time"
  value="<%= now.getTime() + 432000000 %>" />
<fmt:message key="ShipDate"/>
<fmt:formatDate value="${now}" type="date"
  dateStyle="full"/>.
```

SQL Tags

The JSTL SQL tags are designed for quick prototyping and simple applications. For production applications, database operations are normally encapsulated in JavaBeans components.

Table 6–7 SQL Tags

Area	Function	Tags
Database		<code>setDataSource</code>
	SQL	<code>query</code> <code>dateParam</code> <code>param</code> <code>transaction</code> <code>update</code> <code>dateParam</code> <code>param</code>

The `setDataSource` tag is provided to allow you to set data source information for the database. You can provide a JNDI name or `DriverManager` parameters to set the data source information. All the Duke’s Bookstore pages that have more than one SQL tag use the following statement to set the data source:

```
<sql:setDataSource dataSource="jdbc/BookDB" />
```


The query tag is used to perform an SQL query that returns a result set. For parameterized SQL queries, you use a nested param tag inside the query tag.

In `catalog.jsp`, the value of the Add request parameter determines which book information should be retrieved from the database. This parameter is saved as the attribute name `bid` and passed to the param tag. Notice that the query tag obtains its data source from the context attribute `bookDS` set in the context listener.

```
<c:set var="bid" value="${param.Add}"/>
<sql:query var="books" >
    select * from PUBLIC.books where id = ?
    <sql:param value="${bid}" />
</sql:query>
```

The update tag is used to update a database row. The transaction tag is used to perform a series of SQL statements atomically.

The JSP page `receipt.jsp` page uses both tags to update the database inventory for each purchase. Since a shopping cart can contain more than one book, the transaction tag is used to wrap multiple queries and updates. First the page establishes that there is sufficient inventory, then the updates are performed.

```
<c:set var="sufficientInventory" value="true" />
<sql:transaction>
    <c:forEach var="item" items="${sessionScope.cart.items}">
        <c:set var="book" value="${item.item}" />
        <c:set var="bookId" value="${book.bookId}" />

        <sql:query var="books"
            sql="select * from PUBLIC.books where id = ?" >
            <sql:param value="${bookId}" />
        </sql:query>
        <jsp:useBean id="inventory"
            class="database.BookInventory" />
        <c:forEach var="bookRow" begin="0"
            items="${books.rowsByIndex}">
            <jsp:useBean id="bookRow" type="java.lang.Object[]" />
            <jsp:setProperty name="inventory" property="quantity"
                value="<%= (Integer)bookRow[7]%>" />

            <c:if test="${item.quantity > inventory.quantity}">
                <c:set var="sufficientInventory" value="false" />
                <h3><font color="red" size="+2">
                    <fmt:message key="OrderError"/>
                    There is insufficient inventory for
                    <i><c:out value="${bookRow[3]}" /></i>.</font></h3>
            </c:if>
```

```

    </c:forEach>
</c:forEach>

<c:if test="${sufficientInventory == 'true'}" />
  <c:forEach var="item" items="${sessionScope.cart.items}">
    <c:set var="book" value="${item.item}" />
    <c:set var="bookId" value="${book.bookId}" />

    <sql:query var="books"
      sql="select * from PUBLIC.books where id = ?" >
      <sql:param value="${bookId}" />
    </sql:query>

    <c:forEach var="bookRow" begin="0"
      items="${books.rows}">
      <sql:update var="books" sql="update PUBLIC.books set
        inventory = inventory - ? where id = ?" >
        <sql:param value="${item.quantity}" />
        <sql:param value="${bookId}" />
      </sql:update>
    </c:forEach>
  </c:forEach>
  <h3><fmt:message key="ThankYou"/>
  <c:out value="${param.cardname}" />.</h3><br>
</c:if>
</sql:transaction>

```

query Tag Result Interface

The `Result` interface is used to retrieve information from objects returned from a query tag.

```

public interface Result {
    public String[] getColumnNames();
    public int getRowCount();
    public Map[] getRows();
    public Object[][] getRowsByIndex();
    public boolean isLimitedByMaxRows();
}

```

For complete information about this interface, see the API documentation for the `javax.servlet.jsp.jstl.sql` package.

The `var` attribute set by a query tag is of type `Result`. The `getRows` method returns an array of maps that can be supplied to the `items` attribute of a `forEach` tag. The JSTL expression language converts the syntax `${result.rows}` to a

call to `result.getRows`. The expression `${books.rows}` in the following example returns an array of maps.

When you provide a array of maps to the `forEach` tag, the `var` attribute set by the tag is of type `Map`. To retrieve information from a row, use the `get("colname")` method to get a column value. The JSTL expression language converts the syntax `${map.colname}` to a call to `map.get("colname")`. For example, the expression `${book.title}` returns the value of the title entry of a book map.

The Duke's Bookstore page `bookdetails.jsp` retrieves the column values from the book map as follows.

```
<c:forEach var="book" begin="0" items="${books.rows}">
  <h2><c:out value="${book.title}"/></h2>
  &nbsp;<fmt:message key="By"/> <em><c:out
    value="${book.firstname}"/> <c:out
    value="${book.surname}"/></em>&nbsp;&nbsp;&nbsp;
    (<c:out value="${book.year}"/>)<br> &nbsp;&nbsp;&nbsp; <br>
  <h4><fmt:message key="Critics"/></h4>
  <blockquote><c:out value="${book.description}"/>
</blockquote>
  <h4><fmt:message key="ItemPrice"/>:
  <fmt:formatNumber value="${book.price}" type="currency"/>
</h4>
</c:forEach>
```

The following excerpt from `catalog.jsp` uses the `Row` interface to retrieve values from the columns of a book row using scripting language expressions. First the book row that matches a request parameter (`bid`) is retrieved from the database. Since the `bid` and `bookRow` objects are later used by tags that use scripting language expressions to set attribute values and a scriptlet that adds a book to the shopping cart, both objects are declared as scripting variables using the `jsp:useBean` tag. The page creates a bean that describes the book and scripting language expressions are used to set the book properties from book row column values. Finally the book is added to the shopping cart.

You might want to compare this version of `catalog.jsp` to the versions in *JavaServer Pages Technology* (page 83) and *Custom Tags in JSP Pages* (page 113) that use a book database JavaBeans component.

```
<sql:query var="books"
  dataSource="${applicationScope.bookDS}">
  select * from PUBLIC.books where id = ?
<sql:param value="${bid}" />
```

```

</sql:query>
<c:forEach var="bookRow" begin="0"
    items="${books.rowsByIndex}">
    <jsp:useBean id="bid" type="java.lang.String" />
    <jsp:useBean id="bookRow" type="java.lang.Object[]" />
    <jsp:useBean id="addedBook" class="database.BookDetails"
        scope="page" />
    <jsp:setProperty name="addedBook" property="bookId"
        value="<%=bookRow[0]%>" />
    <jsp:setProperty name="addedBook" property="surname"
        value="<%=bookRow[1]%>" />
    <jsp:setProperty name="addedBook" property="firstName"
        value="<%=bookRow[2]%>" />
    <jsp:setProperty name="addedBook" property="title"
        value="<%=bookRow[3]%>" />
    <jsp:setProperty name="addedBook" property="price"
        value="<%=((Double)bookRow[4]).floatValue()%>" />
    <jsp:setProperty name="addedBook" property="year"
        value="<%=((Integer)bookRow[5])%>" />
    <jsp:setProperty name="addedBook"
        property="description" value="<%=bookRow[6]%>" />
    <jsp:setProperty name="addedBook" property="inventory"
        value="<%=((Integer)bookRow[7])%>" />
    </jsp:useBean>
    <% cart.add(bid, addedBook); %>
    ...
</c:forEach>

```

Further Information

For further information on JSTL see:

- The JSTL samples shipped with Sun ONE Application Server 7. The samples are in the directory `<S1AS7_HOME>/samples/webapps/jstl`.
- Resources listed on the Web site <http://java.sun.com/products/jsp/jstl>.
- Reference documentation at <http://jakarta.apache.org/taglibs/doc/standard-doc/standard/index.html>.
- The JSTL 1.0 Specification.

Understanding XML

Eric Armstrong

THIS chapter describes the Extensible Markup Language (XML) and its related specifications. It also gives you practice in writing XML data, so you become comfortably familiar with XML syntax.

Note: The XML files mentioned in this chapter can be found in `<J2EE_HOME>/doc/tutorial/examples/xml/samples`.

Introduction to XML

This section covers the basics of XML. The goal is to give you just enough information to get started, so you understand what XML is all about. (You'll learn about XML in later sections of the tutorial.) We then outline the major features that make XML great for information storage and interchange, and give you a general idea of how XML can be used.

What Is XML?

XML is a text-based markup language that is fast becoming the standard for data interchange on the Web. As with HTML, you identify data using tags (identifiers enclosed in angle brackets, like this: `< . . >`). Collectively, the tags are known as “markup”.

But unlike HTML, XML tags *identify* the data, rather than specifying how to display it. Where an HTML tag says something like “display this data in bold font” (`...`), an XML tag acts like a field name in your program. It puts a label on a piece of data that identifies it (for example: `<message>...</message>`).

Note: Since identifying the data gives you some sense of what *means* (how to interpret it, what you should do with it), XML is sometimes described as a mechanism for specifying the *semantics* (meaning) of the data.

In the same way that you define the field names for a data structure, you are free to use any XML tags that make sense for a given application. Naturally, though, for multiple applications to use the same XML data, they have to agree on the tag names they intend to use.

Here is an example of some XML data you might use for a messaging application:

```
<message>
  <to>you@yourAddress.com</to>
  <from>me@myAddress.com</from>
  <subject>XML Is Really Cool</subject>
  <text>
    How many ways is XML cool? Let me count the ways...
  </text>
</message>
```

Note: Throughout this tutorial, we use boldface text to highlight things we want to bring to your attention. XML does not require anything to be in bold!

The tags in this example identify the message as a whole, the destination and sender addresses, the subject, and the text of the message. As in HTML, the `<to>` tag has a matching end tag: `</to>`. The data between the tag and its matching end tag defines an element of the XML data. Note, too, that the content of the `<to>` tag is entirely contained within the scope of the `<message>...</message>` tag. It is this ability for one tag to contain others that gives XML its ability to represent hierarchical data structures.

Once again, as with HTML, whitespace is essentially irrelevant, so you can format the data for readability and yet still process it easily with a program. Unlike HTML, however, in XML you could easily search a data set for messages con-

taining “cool” in the subject, because the XML tags identify the content of the data, rather than specifying its representation.

Tags and Attributes

Tags can also contain attributes—additional information included as part of the tag itself, within the tag’s angle brackets. The following example shows an email message structure that uses attributes for the “to”, “from”, and “subject” fields:

```
<message to="you@yourAddress.com" from="me@myAddress.com"
  subject="XML Is Really Cool">
  <text>
    How many ways is XML cool? Let me count the ways...
  </text>
</message>
```

As in HTML, the attribute name is followed by an equal sign and the attribute value, and multiple attributes are separated by spaces. Unlike HTML, however, in XML commas between attributes are not ignored—if present, they generate an error.

Since you could design a data structure like `<message>` equally well using either attributes or tags, it can take a considerable amount of thought to figure out which design is best for your purposes. Designing an XML Data Structure (page 233), includes ideas to help you decide when to use attributes and when to use tags.

Empty Tags

One really big difference between XML and HTML is that an XML document is always constrained to be well formed. There are several rules that determine when a document is well-formed, but one of the most important is that every tag has a closing tag. So, in XML, the `</to>` tag is not optional. The `<to>` element is never terminated by any tag other than `</to>`.

Note: Another important aspect of a well-formed document is that all tags are completely nested. So you can have `<message>...<to>...</to>...</message>`, but never `<message>...<to>...</message>...</to>`. A complete list of requirements is contained in the list of XML Frequently Asked Questions (FAQ) at

<http://www.ucc.ie/xml/#FAQ-VALIDWF>. (This FAQ is on the w3c “Recommended Reading” list at <http://www.w3.org/XML/>.)

Sometimes, though, it makes sense to have a tag that stands by itself. For example, you might want to add a “flag” tag that marks message as important. A tag like that doesn’t enclose any content, so it’s known as an “empty tag”. You can create an empty tag by ending it with `</>` instead of `>`. For example, the following message contains such a tag:

```
<message to="you@yourAddress.com" from="me@myAddress.com"
  subject="XML Is Really Cool">
  <flag/>
  <text>
    How many ways is XML cool? Let me count the ways...
  </text>
</message>
```

Note: The empty tag saves you from having to code `<flag></flag>` in order to have a well-formed document. You can control which tags are allowed to be empty by creating a Document Type Definition, or DTD. We’ll talk about that in a few moments. If there is no DTD, then the document can contain any kinds of tags you want, as long as the document is well-formed.

Comments in XML Files

XML comments look just like HTML comments:

```
<message to="you@yourAddress.com" from="me@myAddress.com"
  subject="XML Is Really Cool">
  <!-- This is a comment -->
  <text>
    How many ways is XML cool? Let me count the ways...
  </text>
</message>
```


The XML Prolog

To complete this journeyman's introduction to XML, note that an XML file always starts with a prolog. The minimal prolog contains a declaration that identifies the document as an XML document, like this:

```
<?xml version="1.0"?>
```

The declaration may also contain additional information, like this:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
```

The XML declaration is essentially the same as the HTML header, `<html>`, except that it uses `<?..?>` and it may contain the following attributes:

version

Identifies the version of the XML markup language used in the data. This attribute is not optional.

encoding

Identifies the character set used to encode the data. "ISO-8859-1" is "Latin-1" the Western European and English language character set. (The default is compressed Unicode: UTF-8.)

standalone

Tells whether or not this document references an external entity or an external data type specification (see below). If there are no external references, then "yes" is appropriate

The prolog can also contain definitions of entities (items that are inserted when you reference them from within the document) and specifications that tell which tags are valid in the document, both declared in a Document Type Definition (DTD) that can be defined directly within the prolog, as well as with pointers to external specification files. But those are the subject of later tutorials. For more information on these and many other aspects of XML, see the Recommended Reading list of the w3c XML page at <http://www.w3.org/XML/>.

Note: The declaration is actually optional. But it's a good idea to include it whenever you create an XML file. The declaration should have the version number, at a minimum, and ideally the encoding as well. That standard simplifies things if the XML standard is extended in the future, and if the data ever needs to be localized for different geographical regions.

Everything that comes after the XML prolog constitutes the document's *content*.

Processing Instructions

An XML file can also contain *processing instructions* that give commands or information to an application that is processing the XML data. Processing instructions have the following format:

`<?target instructions?>`

where the *target* is the name of the application that is expected to do the processing, and *instructions* is a string of characters that embodies the information or commands for the application to process.

Since the instructions are application specific, an XML file could have multiple processing instructions that tell different applications to do similar things, though in different ways. The XML file for a slideshow, for example, could have processing instructions that let the speaker specify a technical or executive-level version of the presentation. If multiple presentation programs were used, the program might need multiple versions of the processing instructions (although it would be nicer if such applications recognized standard instructions).

Note: The target name “xml” (in any combination of upper or lowercase letters) is reserved for XML standards. In one sense, the declaration is a processing instruction that fits that standard. (However, when you’re working with the parser later, you’ll see that the method for handling processing instructions never sees the declaration.)

Why Is XML Important?

There are a number of reasons for XML’s surging acceptance. This section lists a few of the most prominent.

Plain Text

Since XML is not a binary format, you can create and edit files with anything from a standard text editor to a visual development environment. That makes it easy to debug your programs, and makes it useful for storing small amounts of data. At the other end of the spectrum, an XML front end to a database makes it possible to efficiently store large amounts of XML data as well. So XML provides scalability for anything from small configuration files to a company-wide data repository.

Data Identification

XML tells you what kind of data you have, not how to display it. Because the markup tags identify the information and break up the data into parts, an email program can process it, a search program can look for messages sent to particular people, and an address book can extract the address information from the rest of the message. In short, because the different parts of the information have been identified, they can be used in different ways by different applications.

Stylability

When display is important, the stylesheet standard, XSL (page 191), lets you dictate how to portray the data. For example, the stylesheet for:

```
<to>you@yourAddress.com</to>
```

can say:

1. Start a new line.
2. Display “To:” in bold, followed by a space
3. Display the destination data.

Which produces:

To: you@yourAddress

Of course, you could have done the same thing in HTML, but you wouldn’t be able to process the data with search programs and address-extraction programs and the like. More importantly, since XML is inherently style-free, you can use a completely different stylesheet to produce output in postscript, TEX, PDF, or some new format that hasn’t even been invented yet. That flexibility amounts to what one author described as “future-proofing” your information. The XML documents you author today can be used in future document-delivery systems that haven’t even been imagined yet.

Inline Reusability

One of the nicer aspects of XML documents is that they can be composed from separate entities. You can do that with HTML, but only by linking to other documents. Unlike HTML, XML entities can be included “in line” in a document. The included sections look like a normal part of the document—you can search

the whole document at one time or download it in one piece. That lets you modularize your documents without resorting to links. You can single-source a section so that an edit to it is reflected everywhere the section is used, and yet a document composed from such pieces looks for all the world like a one-piece document.

Linkability

Thanks to HTML, the ability to define links between documents is now regarded as a necessity. The next section of this tutorial, *XML and Related Specs: Digesting the Alphabet Soup* (page 187), discusses the link-specification initiative. This initiative lets you define two-way links, multiple-target links, “expanding” links (where clicking a link causes the targeted information to appear inline), and links between two existing documents that are defined in a third.

Easily Processed

As mentioned earlier, regular and consistent notation makes it easier to build a program to process XML data. For example, in HTML a `<dt>` tag can be delimited by `</dt>`, another `<dt>`, `<dd>`, or `</dl>`. That makes for some difficult programming. But in XML, the `<dt>` tag must always have a `</dt>` terminator, or else it will be defined as a `<dt/>` tag. That restriction is a critical part of the constraints that make an XML document well-formed. (Otherwise, the XML parser won’t be able to read the data.) And since XML is a vendor-neutral standard, you can choose among several XML parsers, any one of which takes the work out of processing XML data.

Hierarchical

Finally, XML documents benefit from their hierarchical structure. Hierarchical document structures are, in general, faster to access because you can drill down to the part you need, like stepping through a table of contents. They are also easier to rearrange, because each piece is delimited. In a document, for example, you could move a heading to a new location and drag everything under it along with the heading, instead of having to page down to make a selection, cut, and then paste the selection into a new location.

How Can You Use XML?

There are several basic ways to make use of XML:

- Traditional data processing, where XML encodes the data for a program to process
- Document-driven programming, where XML documents are containers that build interfaces and applications from existing components
- Archiving—the foundation for document-driven programming, where the customized version of a component is saved (archived) so it can be used later
- Binding, where the DTD or schema that defines an XML data structure is used to automatically generate a significant portion of the application that will eventually process that data

Traditional Data Processing

XML is fast becoming the data representation of choice for the Web. It's terrific when used in conjunction with network-centric Java-platform programs that send and retrieve information. So a client/server application, for example, could transmit XML-encoded data back and forth between the client and the server.

In the future, XML is potentially the answer for data interchange in all sorts of transactions, as long as both sides agree on the markup to use. (For example, should an e-mail program expect to see tags named <FIRST> and <LAST>, or <FIRSTNAME> and <LASTNAME>?) The need for common standards will generate a lot of industry-specific standardization efforts in the years ahead. In the meantime, mechanisms that let you “translate” the tags in an XML document will be important. Such mechanisms include projects like the RDF (page 196) initiative, which defines “metatags”, and the XSL (page 191) specification, which lets you translate XML tags into other XML tags.

Document-Driven Programming (DDP)

The newest approach to using XML is to construct a document that describes how an application page should look. The document, rather than simply being displayed, consists of references to user interface components and business-logic components that are “hooked together” to create an application on the fly.

Of course, it makes sense to utilize the Java platform for such components. Both Java BeansTM for interfaces and Enterprise Java BeansTM for business logic can

be used to construct such applications. Although none of the efforts undertaken so far are ready for commercial use, much preliminary work has already been done.

Note: The Java programming language is also excellent for writing XML-processing tools that are as portable as XML. Several Visual XML editors have been written for the Java platform. For a listing of editors, processing tools, and other XML resources, see the “Software” section of Robin Cover’s SGML/XML Web Page at <http://www.oasis-open.org/cover/>.

Binding

Once you have defined the structure of XML data using either a DTD or the one of the schema standards, a large part of the processing you need to do has already been defined. For example, if the schema says that the text data in a `<date>` element must follow one of the recognized date formats, then one aspect of the validation criteria for the data has been defined—it only remains to write the code. Although a DTD specification cannot go the same level of detail, a DTD (like a schema) provides a grammar that tells which data structures can occur, in what sequences. That specification tells you how to write the high-level code that processes the data elements.

But when the data structure (and possibly format) is fully specified, the code you need to process it can just as easily be generated automatically. That process is known as *binding*—creating classes that recognize and process different data elements by processing the specification that defines those elements. As time goes on, you should find that you are using the data specification to generate significant chunks of code, so you can focus on the programming that is unique to your application.

Archiving

The Holy Grail of programming is the construction of reusable, modular components. Ideally, you’d like to take them off the shelf, customize them, and plug them together to construct an application, with a bare minimum of additional coding and additional compilation.

The basic mechanism for saving information is called *archiving*. You archive a component by writing it to an output stream in a form that you can reuse later. You can then read it in and instantiate it using its saved parameters. (For exam-

ple, if you saved a table component, its parameters might be the number of rows and columns to display.) Archived components can also be shuffled around the Web and used in a variety of ways.

When components are archived in binary form, however, there are some limitations on the kinds of changes you can make to the underlying classes if you want to retain compatibility with previously saved versions. If you could modify the archived version to reflect the change, that would solve the problem. But that's hard to do with a binary object. Such considerations have prompted a number of investigations into using XML for archiving. But if an object's state were archived in text form using XML, then anything and everything in it could be changed as easily as you can say, "search and replace".

XML's text-based format could also make it easier to transfer objects between applications written in different languages. For all of these reasons, XML-based archiving is likely to become an important force in the not-too-distant future.

Summary

XML is pretty simple, and very flexible. It has many uses yet to be discovered—we are just beginning to scratch the surface of its potential. It is the foundation for a great many standards yet to come, providing a common language that different computer systems can use to exchange data with one another. As each industry-group comes up with standards for what they want to say, computers will begin to link to each other in ways previously unimaginable.

For more information on the background and motivation of XML, see this great article in Scientific American at

<http://www.sciam.com/1999/0599issue/0599bosak.html>

XML and Related Specs: Digesting the Alphabet Soup

Now that you have a basic understanding of XML, it makes sense to get a high-level overview of the various XML-related acronyms and what they mean. There is a lot of work going on around XML, so there is a lot to learn.

The current APIs for accessing XML documents either serially or in random access mode are, respectively, SAX (page 189) and DOM (page 189). The specifications for ensuring the validity of XML documents are DTD (page 190) (the

original mechanism, defined as part of the XML specification) and various Schema Standards (page 192) proposals (newer mechanisms that use XML syntax to do the job of describing validation criteria).

Other future standards that are nearing completion include the XSL (page 191) standard—a mechanism for setting up translations of XML documents (for example to HTML or other XML) and for dictating how the document is rendered. The transformation part of that standard, XSLT (+XPath) (page 191), is completed and covered in this tutorial. Another effort nearing completion is the XML Link Language specification (XML Linking, page 194), which enables links between XML documents.

Those are the major initiatives you will want to be familiar with. This section also surveys a number of other interesting proposals, including the HTML-lookalike standard, XHTML (page 195), and the meta-standard for describing the information an XML document contains, RDF (page 196). There are also standards efforts that extend XML's capabilities, such as XLink and XPointer.

Finally, there are a number of interesting standards and standards-proposals that build on XML, including Synchronized Multimedia Integration Language (SMIL, page 197), Mathematical Markup Language (MathML, page 197), Scalable Vector Graphics (SVG, page 197), and DrawML (page 198), as well as a number of eCommerce standards.

The remainder of this section gives you a more detailed description of these initiatives. To help keep things straight, it's divided into:

- Basic Standards (page 188)
- Schema Standards (page 192)
- Linking and Presentation Standards (page 194)
- Knowledge Standards (page 196)
- Standards That Build on XML (page 197)

Skim the terms once, so you know what's here, and keep a copy of this document handy so you can refer to it whenever you see one of these terms in something you're reading. Pretty soon, you'll have them all committed to memory, and you'll be at least "conversant" with XML!

Basic Standards

These are the basic standards you need to be familiar with. They come up in pretty much any discussion of XML.

SAX

Simple API for XML

This API was actually a product of collaboration on the XML-DEV mailing list, rather than a product of the W3C. It's included here because it has the same “final” characteristics as a W3C recommendation.

You can also think of this standard as the “serial access” protocol for XML. This is the fast-to-execute mechanism you would use to read and write XML data in a server, for example. This is also called an event-driven protocol, because the technique is to register your handler with a SAX parser, after which the parser invokes your callback methods whenever it sees a new XML tag (or encounters an error, or wants to tell you anything else).

DOM

Document Object Model

The Document Object Model protocol converts an XML document into a collection of objects in your program. You can then manipulate the object model in any way that makes sense. This mechanism is also known as the “random access” protocol, because you can visit any part of the data at any time. You can then modify the data, remove it, or insert new data.

JDOM and dom4j

While the Document Object Model (DOM) provides a lot of power for document-oriented processing, it doesn't provide much in the way of object-oriented simplification. Java developers who are processing more data-oriented structures—rather than books, articles, and other full-fledged documents—frequently find that object-oriented APIs like JDOM and dom4j are easier to use and more suited to their needs.

Here are the important differences to understand when choosing between the two:

- JDOM is somewhat cleaner, smaller API. Where “coding style” is an important consideration, JDOM is a good choice.
- JDOM is a Java Community Process (JCP) initiative. When completed, it will be an endorsed standard.
- dom4j is a smaller, faster implementation that has been in wide use for a number of years.
- dom4j is a factory-based implementation. That makes it easier to modify for complex, special-purpose applications. At the time of this writing, JDOM does not yet use a factory to instantiate an instance of the parser (although the standard appears to be headed in that direction). So, with JDOM, you always get the original parser. (That’s fine for the majority of applications, but may not be appropriate if your application has special needs.)

For more information on JDOM, see <http://www.jdom.org/>.

For more information on dom4j, see <http://dom4j.org/>.

DTD

Document Type Definition

The DTD specification is actually part of the XML specification, rather than a separate entity. On the other hand, it is optional—you can write an XML document without it. And there are a number of Schema Standards (page 192) proposals that offer more flexible alternatives. So it is treated here as though it were a separate specification.

A DTD specifies the kinds of tags that can be included in your XML document, and the valid arrangements of those tags. You can use the DTD to make sure you don’t create an invalid XML structure. You can also use it to make sure that the XML structure you are reading (or that got sent over the net) is indeed valid.

Unfortunately, it is difficult to specify a DTD for a complex document in such a way that it prevents all invalid combinations and allows all the valid ones. So constructing a DTD is something of an art. The DTD can exist at the front of the document, as part of the prolog. It can also exist as a separate entity, or it can be split between the document prolog and one or more additional entities.

However, while the DTD mechanism was the first method defined for specifying valid document structure, it was not the last. Several newer schema specifications have been devised. You'll learn about those momentarily.

Namespaces

The namespace standard lets you write an XML document that uses two or more sets of XML tags in modular fashion. Suppose for example that you created an XML-based parts list that uses XML descriptions of parts supplied by other manufacturers (online!). The “price” data supplied by the subcomponents would be amounts you want to total up, while the “price” data for the structure as a whole would be something you want to display. The namespace specification defines mechanisms for qualifying the names so as to eliminate ambiguity. That lets you write programs that use information from other sources and do the right things with it.

The latest information on namespaces can be found at <http://www.w3.org/TR/REC-xml-names>.

XSL

Extensible Stylesheet Language

The XML standard specifies how to identify data, not how to display it. HTML, on the other hand, told how things should be displayed without identifying what they were. The XSL standard has two parts, XSLT (the transformation standard, described next) and XSL-FO (the part that covers *formatting objects*, also known as *flow objects*). XSL-FO gives you the ability to define multiple areas on a page and then link them together. When a text stream is directed at the collection, it fills the first area and then “flows” into the second when the first area is filled. Such objects are used by newsletters, catalogs, and periodical publications.

The latest W3C work on XSL is at <http://www.w3.org/TR/WD-xsl>.

XSLT (+XPATH)

Extensible Stylesheet Language for Transformations

The XSLT transformation standard is essentially a translation mechanism that lets you specify what to convert an XML tag into so that it can be displayed—for example, in HTML. Different XSL formats can then be used to display the same data in different ways, for different uses. (The XPATH standard is an addressing

mechanism that you use when constructing transformation instructions, in order to specify the parts of the XML structure you want to transform.)

Schema Standards

A DTD makes it possible to validate the structure of relatively simple XML documents, but that's as far as it goes.

A DTD can't restrict the content of elements, and it can't specify complex relationships. For example, it is impossible to specify with a DTD that a `<heading>` for a `<book>` must have both a `<title>` and an `<author>`, while a `<heading>` for a `<chapter>` only needs a `<title>`. In a DTD, once you only get to specify the structure of the `<heading>` element one time. There is no context-sensitivity.

This issue stems from the fact that a DTD specification is not hierarchical. For a mailing address that contained several "parsed character data" (PCDATA) elements, for example, the DTD might look something like this:

```
<!ELEMENT mailAddress (name, address, zipcode)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT address (#PCDATA)>
<!ELEMENT zipcode (#PCDATA)>
```

As you can see, the specifications are linear. That fact forces you to come up with new names for similar elements in different settings. So if you wanted to add another "name" element to the DTD that contained the `<firstName>`, `<middleInitial>`, and `<lastName>`, then you would have to come up with another identifier. You could not simply call it "name" without conflicting with the `<name>` element defined for use in a `<mailAddress>`.

Another problem with the non hierarchical nature of DTD specifications is that it is not clear what comments are meant to explain. A comment at the top like `<!-- Address used for mailing via the postal system -->` would apply to all of the elements that constitute a mailing address. But a comment like `<!-- Addressee -->` would apply to the name element only. On the other hand, a comment like `<!-- A 5-digit string -->` would apply specifically to the #PCDATA part of the zipcode element, to describe the valid formats. Finally, DTDs do not allow you to formally specify field-validation criteria, such as the 5-digit (or 5 and 4) limitation for the zipcode field.

Finally, a DTD uses syntax which substantially different from XML, so it can't be processed with a standard XML parser. That means you can't read a DTD into a DOM, for example, modify it, and then write it back out again.

To remedy these shortcomings, a number of proposals have been made for a more database-like, hierarchical “schema” that specifies validation criteria. The major proposals are shown below.

XML Schema

A large, complex standard that has two parts. One part specifies structure relationships. (This is the largest and most complex part.) The other part specifies mechanisms for validating the content of XML elements by specifying a (potentially very sophisticated) *datatype* for each element. The good news is that XML Schema for Structures lets you specify any kind of relationship you can conceive of. The bad news is that it takes a lot of work to implement, and it takes a bit of learning to use. Most of the alternatives provide for simpler structure definitions, while incorporating the XML Schema datatype standard.

For more information on the XML Schema, see the W3C specs XML Schema (Structures) and XML Schema (Datatypes), as well as other information accessible at <http://www.w3c.org/XML/Schema>.

RELAX NG

Regular Language description for XML

Simpler than XML Structure Schema, is an emerging standard under the auspices of OASIS (Organization for the Advancement of Structured Information Systems). RELAX NG use regular expression patterns to express constraints on structure relationships, and it is designed to work with the XML Schema datatyping mechanism to express content constraints. This standard also uses XML syntax, and it includes a DTD to RELAX converter. (“NG” stands for “Next Generation”. It’s a newer version of the RELAX schema mechanism that integrates TREX.)

For more information on RELAX NG, see <http://www.oasis-open.org/committees/relax-ng/>

TREX

Tree Regular Expressions for XML

A means of expressing validation criteria by describing a *pattern* for the structure and content of an XML document. Now part of the RELAX NG specification.

For more information on TREX, see <http://www.thaiopensource.com/trex/>.

SOX

Schema for Object-oriented XML

SOX is a schema proposal that includes extensible data types, namespaces, and embedded documentation.

For more information on SOX, see <http://www.w3.org/TR/NOTE-SOX>.

Schematron

Schema for Object-oriented XML

An assertion-based schema mechanism that allows for sophisticated validation.

For more information on the Schematron validation mechanism, see <http://www.ascc.net/xml/resource/schematron/schematron.html>.

Linking and Presentation Standards

Arguably the two greatest benefits provided by HTML were the ability to link between documents, and the ability to create simple formatted documents (and, eventually, very complex formatted documents). The following standards aim at preserving the benefits of HTML in the XML arena, and to adding additional functionality, as well.

XML Linking

These specifications provide a variety of powerful linking mechanisms, and are sure to have a big impact on how XML documents are used.

XLink

The XLink protocol is a specification for handling links between XML documents. This specification allows for some pretty sophisticated linking, including two-way links, links to multiple documents, “expanding” links that insert the linked information into your document rather than replacing your document with a new page, links between two documents that are created in a third, independent document, and indirect links (so you can point to

an “address book” rather than directly to the target document—updating the address book then automatically changes any links that use it).

XML Base

This standard defines an attribute for XML documents that defines a “base” address, that is used when evaluating a relative address specified in the document. (So, for example, a simple file name would be found in the base-address directory.)

XPointer

In general, the XLink specification targets a document or document-segment using its ID. The XPointer specification defines mechanisms for “addressing into the internal structures of XML documents”, without requiring the author of the document to have defined an ID for that segment. To quote the spec, it provides for “reference to elements, character strings, and other parts of XML documents, whether or not they bear an explicit ID attribute”.

For more information on the XML Linking standards, see <http://www.w3.org/XML/Linking>.

XHTML

The XHTML specification is a way of making XML documents that look and act like HTML documents. Since an XML document can contain any tags you care to define, why not define a set of tags that look like HTML? That’s the thinking behind the XHTML specification, at any rate. The result of this specification is a document that can be displayed in browsers and also treated as XML data. The data may not be quite as identifiable as “pure” XML, but it will be a heck of a lot easier to manipulate than standard HTML, because XML specifies a good deal more regularity and consistency.

For example, every tag in a well-formed XML document must either have an end-tag associated with it or it must end in `</>`. So you might see `<p> . . . </p>`, or you might see `<p/>`, but you will never see `<p>` standing by itself. The upshot of that requirement is that you never have to program for the weird kinds of cases you see in HTML where, for example, a `<dt>` tag might be terminated by `</DT>`, by another `<DT>`, by `<dd>`, or by `</dl>`. That makes it a lot easier to write code!

The XHTML specification is a reformulation of HTML 4.0 into XML. The latest information is at <http://www.w3.org/TR/xhtml1>.

Knowledge Standards

When you start looking down the road five or six years, and visualize how the information on the Web will begin to turn into one huge knowledge base (the “semantic Web”). For the latest on the semantic Web, visit <http://www.w3.org/2001/sw/>.

In the meantime, here are the fundamental standards you’ll want to know about:

RDF

Resource Description Framework

RDF is a standard for defining *meta* data -- information that describes what a particular data item is, and specifies how it can be used. Used in conjunction with the XHTML specification, for example, or with HTML pages, RDF could be used to describe the content of the pages. For example, if your browser stored your ID information as FIRSTNAME, LASTNAME, and EMAIL, an RDF description could make it possible to transfer data to an application that wanted NAME and EMAILADDRESS. Just think: One day you may not need to type your name and address at every Web site you visit!

For the latest information on RDF, see <http://www.w3.org/TR/REC-rdf-syntax>.

RDF Schema

RDF Schema allows the specification of consistency rules and additional information that describe how the statements in a Resource Description Framework (RDF) should be interpreted.

For more information on the RDF Schema recommendation, see <http://www.w3.org/TR/rdf-schema>.

XTM

XML Topic Maps

In many ways a simpler, more readily usable knowledge-representation than RDF, the topic maps standard is one worth watching. So far, RDF is the W3C standard for knowledge representation, but topic maps could possibly become the “developer’s choice” among knowledge representation standards.

For more information on XML Topic Maps, <http://www.topic-maps.org/xtm/index.html>. For information on topic maps and the Web, see <http://www.topicmaps.org/>.

Standards That Build on XML

The following standards and proposals build on XML. Since XML is basically a language-definition tool, these specifications use it to define standardized languages for specialized purposes.

Extended Document Standards

These standards define mechanisms for producing extremely complex documents—books, journals, magazines, and the like—using XML.

SMIL

Synchronized Multimedia Integration Language

SMIL is a W3C recommendation that covers audio, video, and animations. It also addresses the difficult issue of synchronizing the playback of such elements.

For more information on SMIL, see <http://www.w3.org/TR/REC-smil>.

MathML

Mathematical Markup Language

MathML is a W3C recommendation that deals with the representation of mathematical formulas.

For more information on MathML, see <http://www.w3.org/TR/REC-MathML>.

SVG

Scalable Vector Graphics

SVG is a W3C working draft that covers the representation of vector graphic images. (Vector graphic images that are built from commands that say things like “draw a line (square, circle) from point *x*₁ to point *m*,*n*” rather than encoding the image as a series of bits. Such images are more easily scalable, although they typically require more processing time to render.)

For more information on SVG, see <http://www.w3.org/TR/WD-SVG>.

DrawML

Drawing Meta Language

DrawML is a W3C note that covers 2D images for technical illustrations. It also addresses the problem of updating and refining such images.

For more information on DrawML, see <http://www.w3.org/TR/NOTE-drawml>.

eCommerce Standards

These standards are aimed at using XML in the world of business-to-business (B2B) and business-to-consumer (B2C) commerce.

ICE

Information and Content Exchange

ICE is a protocol for use by content syndicators and their subscribers. It focuses on “automating content exchange and reuse, both in traditional publishing contexts and in business-to-business relationships”.

For more information on ICE, see <http://www.w3.org/TR/NOTE-ice>.

ebXML

Electronic Business with XML

This standard aims at creating a modular electronic business framework using XML. It is the product of a joint initiative by the United Nations (UN/CEFACT) and the Organization for the Advancement of Structured Information Systems (OASIS).

For more information on ebXML, see <http://www.ebxml.org/>.

cxml

Commerce XML

cxml is a RosettaNet (www.rosettanet.org) standard for setting up interactive online catalogs for different buyers, where the pricing and product offerings are company specific. Includes mechanisms to handle purchase orders, change orders, status updates, and shipping notifications.

For more information on cxml, see <http://www.cxml.org/>

CBL

Common Business Library

CBL is a library of element and attribute definitions maintained by CommerceNet (www.commerce.net).

For more information on CBL and a variety of other initiatives that work together to enable eCommerce applications, see http://www.commerce.net/projects/current-projects/eco/wg/eCo_Framework_Specifications.html.

UBL

Universal Business Language

An OASIS initiative aimed at compiling a standard library of XML business documents (purchase orders, invoices, etc.) that are defined with XML Schema definitions.

For more information on UBL, see <http://www.oasis-open.org/committees/ubl>.

Summary

XML is becoming a widely-adopted standard that is being used in a dizzying variety of application areas.

Generating XML Data

This section also takes you step by step through the process of constructing an XML document. Along the way, you'll gain experience with the XML components you'll typically use to create your data structures.

Writing a Simple XML File

You'll start by writing the kind of XML data you could use for a slide presentation. In this exercise, you'll use your text editor to create the data in order to become comfortable with the basic format of an XML file. You'll be using this file and extending it in later exercises.

Creating the File

Using a standard text editor, create a file called `slideSample.xml`.

Note: Here is a version of it that already exists: `slideSample01.xml`. (The browsable version is `slideSample01-xml.html`.) You can use this version to compare your work, or just review it as you read this guide.

Writing the Declaration

Next, write the declaration, which identifies the file as an XML document. The declaration starts with the characters “<?”, which is the standard XML identifier for a *processing instruction*. (You’ll see other processing instructions later on in this tutorial.)

```
<?xml version='1.0' encoding='utf-8'?>
```

This line identifies the document as an XML document that conforms to version 1.0 of the XML specification, and says that it uses the 8-bit Unicode character-encoding scheme. (For information on encoding schemes, see *Java Encoding Schemes* (page 425).)

Since the document has not been specified as “standalone”, the parser assumes that it may contain references to other documents. To see how to specify a document as “standalone”, see *The XML Prolog* (page 181).

Adding a Comment

Comments are ignored by XML parsers. A program will never see them in fact, unless you activate special settings in the parser. Add the text highlighted below to put a comment into the file.

```
<?xml version='1.0' encoding='utf-8'?>
```

```
<!-- A SAMPLE set of slides -->
```

Defining the Root Element

After the declaration, every XML file defines exactly one element, known as the root element. Any other elements in the file are contained within that element.

Enter the text highlighted below to define the root element for this file, `slide-show`:

```
<?xml version='1.0' encoding='utf-8'?>

<!-- A SAMPLE set of slides -->

<slideshow>

</slideshow>
```

Note: XML element names are case-sensitive. The end-tag must exactly match the start-tag.

Adding Attributes to an Element

A slide presentation has a number of associated data items, none of which require any structure. So it is natural to define them as attributes of the `slide-show` element. Add the text highlighted below to set up some attributes:

```
...
<slideshow
  title="Sample Slide Show"
  date="Date of publication"
  author="Yours Truly"
>
</slideshow>
```

When you create a name for a tag or an attribute, you can use hyphens (“-”), underscores (“_”), colons (“:”), and periods (“.”) in addition to characters and numbers. Unlike HTML, values for XML attributes are always in quotation marks, and multiple attributes are never separated by commas.

Note: Colons should be used with care or avoided altogether, because they are used when defining the namespace for an XML document.

Adding Nested Elements

XML allows for hierarchically structured data, which means that an element can contain other elements. Add the text highlighted below to define a slide element and a title element contained within it:

```
<slideshow
  ...
>

  <!-- TITLE SLIDE -->
  <slide type="all">
    <title>Wake up to WonderWidgets!</title>
  </slide>

</slideshow>
```

Here you have also added a type attribute to the slide. The idea of this attribute is that slides could be earmarked for a mostly technical or mostly executive audience with `type="tech"` or `type="exec"`, or identified as suitable for both with `type="all"`.

More importantly, though, this example illustrates the difference between things that are more usefully defined as elements (the `title` element) and things that are more suitable as attributes (the `type` attribute). The visibility heuristic is primarily at work here. The title is something the audience will see. So it is an element. The type, on the other hand, is something that never gets presented, so it is an attribute. Another way to think about that distinction is that an element is a container, like a bottle. The type is a characteristic of the *container* (is it tall or short, wide or narrow). The title is a characteristic of the *contents* (water, milk, or tea). These are not hard and fast rules, of course, but they can help when you design your own XML structures.

Adding HTML-Style Text

Since XML lets you define any tags you want, it makes sense to define a set of tags that look like HTML. The XHTML standard does exactly that, in fact. You'll see more about that towards the end of the SAX tutorial. For now, type the

text highlighted below to define a slide with a couple of list item entries that use an HTML-style `` tag for emphasis (usually rendered as italicized text):

```
...
<!-- TITLE SLIDE -->
<slide type="all">
  <title>Wake up to WonderWidgets!</title>
</slide>

<!-- OVERVIEW -->
<slide type="all">
  <title>Overview</title>
  <item>Why <em>WonderWidgets</em> are great</item>
  <item>Who <em>buys</em> WonderWidgets</item>
</slide>

</slideshow>
```

Note that defining a *title* element conflicts with the XHTML element that uses the same name. We'll discuss the mechanism that produces the conflict (the DTD), along with possible solutions, later on in this tutorial.

Adding an Empty Element

One major difference between HTML and XML, though, is that all XML must be *well-formed* — which means that every tag must have an ending tag or be an empty tag. You're getting pretty comfortable with ending tags, by now. Add the text highlighted below to define an empty list item element with no contents:

```
...
<!-- OVERVIEW -->
<slide type="all">
  <title>Overview</title>
  <item>Why <em>WonderWidgets</em> are great</item>
  <item/>
  <item>Who <em>buys</em> WonderWidgets</item>
</slide>

</slideshow>
```

Note that any element can be empty element. All it takes is ending the tag with `</>` instead of `</>`. You could do the same thing by entering `<item></item>`, which is equivalent.

Note: Another factor that makes an XML file *well-formed* is proper nesting. So `<i>some_text</i>` is well-formed, because the `<i>...</i>` sequence is completely nested within the `...` tag. This sequence, on the other hand, is not well-formed: `<i>some_text</i>`.

The Finished Product

Here is the completed version of the XML file:

```
<?xml version='1.0' encoding='utf-8'?>

<!-- A SAMPLE set of slides -->

<slideshow
  title="Sample Slide Show"
  date="Date of publication"
  author="Yours Truly"
>

  <!-- TITLE SLIDE -->
  <slide type="all">
    <title>Wake up to WonderWidgets!</title>
  </slide>

  <!-- OVERVIEW -->
  <slide type="all">
    <title>Overview</title>
    <item>Why <em>WonderWidgets</em> are great</item>
    <item/>
    <item>Who <em>buys</em> WonderWidgets</item>
  </slide>
</slideshow>
```

Save a copy of this file as `slideSample01.xml`, so you can use it as the initial data structure when experimenting with XML programming operations.

Writing Processing Instructions

It sometimes makes sense to code application-specific processing instructions in the XML data. In this exercise, you'll add a processing instruction to your `slideSample.xml` file.

Note: The file you'll create in this section is `slideSample02.xml`. (The browsable version is `slideSample02-xml.html`.)

As you saw in Processing Instructions (page 182), the format for a processing instruction is `<?target data?>`, where “target” is the target application that is expected to do the processing, and “data” is the instruction or information for it to process. Add the text highlighted below to add a processing instruction for a mythical slide presentation program that will query the user to find out which slides to display (technical, executive-level, or all):

```
<slideshow
...
>

<!-- PROCESSING INSTRUCTION -->
<?my.presentation.Program QUERY="exec, tech, all"?>

<!-- TITLE SLIDE -->
```

Notes:

- The “data” portion of the processing instruction can contain spaces, or may even be null. But there cannot be any space between the initial `<?` and the target identifier.
- The data begins after the first space.
- Fully qualifying the target with the complete Web-unique package prefix makes sense, so as to preclude any conflict with other programs that might process the same data.
- For readability, it seems like a good idea to include a colon (:) after the name of the application, like this:

```
<?my.presentation.Program: QUERY="..."?>
```

The colon makes the target name into a kind of “label” that identifies the intended recipient of the instruction. However, while the w3c spec allows “:” in a target name, some versions of IE5 consider it an error. For this tutorial, then, we avoid using a colon in the target name.

Save a copy of this file as `slideSample02.xml`, so you can use it when experimenting with processing instructions.

Introducing an Error

The parser can generate one of three kinds of errors: fatal error, error, and warning. In this exercise, you'll make a simple modification to the XML file to introduce a fatal error. Then you'll see how it's handled in the Echo app.

Note: The XML structure you'll create in this exercise is in `slideSampleBad1.xml`. (The browsable version is `slideSampleBad1-xml.html`.)

One easy way to introduce a fatal error is to remove the final `/` from the empty `item` element to create a tag that does not have a corresponding end tag. That constitutes a fatal error, because all XML documents must, by definition, be well formed. Do the following:

1. Copy `slideSample02.xml` to `slideSampleBad1.xml`.
2. Edit `slideSampleBad1.xml` and remove the character shown below:

```
...
<!-- OVERVIEW -->
<slide type="all">
  <title>Overview</title>
  <item>Why <em>WonderWidgets</em> are great</item>
  <item/>
  <item>Who <em>buys</em> WonderWidgets</item>
</slide>
...
```

to produce:

```
...
<item>Why <em>WonderWidgets</em> are great</item>
<item>
<item>Who <em>buys</em> WonderWidgets</item>
...
```

Now you have a file that you can use to generate an error in any parser, any time. (XML parsers are required to generate a fatal error for this file, because the lack of an end-tag for the `<item>` element means that the XML structure is no longer *well-formed*.)

Substituting and Inserting Text

In this section, you'll learn about:

- Handling Special Characters (“<”, “&”, and so on)
- Handling Text with XML-style syntax

Handling Special Characters

In XML, an entity is an XML structure (or plain text) that has a name. Referencing the entity by name causes it to be inserted into the document in place of the entity reference. To create an entity reference, the entity name is surrounded by an ampersand and a semicolon, like this:

```
&entityName;
```

Later, when you learn how to write a DTD, you'll see that you can define your own entities, so that `&yourEntityName;` expands to all the text you defined for that entity. For now, though, we'll focus on the predefined entities and character references that don't require any special definitions.

Predefined Entities

An entity reference like `&` contains a name (in this case, “amp”) between the start and end delimiters. The text it refers to (&) is substituted for the name, like a macro in a programming language. Table 7–1 shows the predefined entities for special characters.

Table 7–1 Predefined Entities

Character	Reference
&	&
<	<
>	>
"	"
'	'

Character References

A character reference like `“` contains a hash mark (#) followed by a number. The number is the Unicode value for a single character, such as 65 for the letter “A”, 147 for the left-curly quote, or 148 for the right-curly quote. In this case, the “name” of the entity is the hash mark followed by the digits that identify the character.

Note: XML expects values to be specified in decimal. However, the Unicode charts at <http://www.unicode.org/charts/> specify values in hexadecimal! So you’ll need to do a conversion to get the right value to insert into your XML data set.

Using an Entity Reference in an XML Document

Suppose you wanted to insert a line like this in your XML document:

Market Size < predicted

The problem with putting that line into an XML file directly is that when the parser sees the left-angle bracket (<), it starts looking for a tag name, which throws off the parse. To get around that problem, you put `<` in the file, instead of “<”.

Note: The results of the modifications below are contained in `slideSample03.xml`.

Add the text highlighted below to your `slideSample.xml` file, and save a copy of it for future use as `slideSample03.xml`:

```
<!-- OVERVIEW -->
<slide type="all">
  <title>Overview</title>
  ...
</slide>

<slide type="exec">
  <title>Financial Forecast</title>
  <item>Market Size &lt; predicted</item>
  <item>Anticipated Penetration</item>
  <item>Expected Revenues</item>
  <item>Profit Margin </item>
</slide>

</slideshow>
```

When you use an XML parser to echo this data, you will see the desired output:

```
Market Size < predicted
```

You see an angle bracket (“<”) where you coded “<”, because the XML parser converts the reference into the entity it represents, and passes that entity to the application.

Handling Text with XML-Style Syntax

When you are handling large blocks of XML or HTML that include many of the special characters, it would be inconvenient to replace each of them with the appropriate entity reference. For those situations, you can use a CDATA section.

Note: The results of the modifications below are contained in `slideSample04.xml`.

A CDATA section works like `<pre>...</pre>` in HTML, only more so—all whitespace in a CDATA section is significant, and characters in it are not interpreted as XML. A CDATA section starts with `<![CDATA[` and ends with `]]>`.

Add the text highlighted below to your `slideSample.xml` file to define a CDATA section for a fictitious technical slide, and save a copy of the file as `slideSample04.xml`:

```
...
<slide type="tech">
  <title>How it Works</title>
  <item>First we fizzle the frobmorten</item>
  <item>Then we framboze the staten</item>
  <item>Finally, we frenzle the fuznaten</item>
  <item><![CDATA[Diagram:
    frobmorten <----- fuznaten
      |      <3>^
      | <1>|   <1> = fizzle
      V     | <2> = framboze
    Staten-----+<3> = frenzle
      <2>
  ]]></item>
</slide>
</slideshow>
```

When you echo this file with an XML parser, you'll see the following output:

```
Diagram:
frobmorten <-----fuznaten
  |      <3>      ^
  | <1>          |   <1> = fizzle
  V             |   <2> = framboze
staten-----+   <3> = frenzle
      <2>
```

The point here is that the text in the CDATA section will have arrived as it was written. Since the parser doesn't treat the angle brackets as XML, they don't generate the fatal errors they would otherwise cause. (Because, if the angle brackets weren't in a CDATA section, the document would not be well-formed.)

Creating a Document Type Definition (DTD)

After the XML declaration, the document prolog can include a DTD, which lets you specify the kinds of tags that can be included in your XML document. In addition to telling a validating parser which tags are valid, and in what arrangements, a DTD tells both validating and nonvalidating parsers where text is

expected, which lets the parser determine whether the whitespace it sees is significant or ignorable.

Basic DTD Definitions

To begin learning about DTD definitions, let's start by telling the parser where text is expected and where any text (other than whitespace) would be an error. (Whitespace in such locations is *ignorable*.)

Note: The DTD defined in this section is contained in `slideshow1a.dtd`. (The browsable version is `slideshow1a-dtd.html`.)

Start by creating a file named `slideshow.dtd`. Enter an XML declaration and a comment to identify the file, as shown below:

```
<?xml version='1.0' encoding='utf-8'?>

<!--
  DTD for a simple "slide show".
-->
```

Next, add the text highlighted below to specify that a `slideshow` element contains `slide` elements and nothing else:

```
<!-- DTD for a simple "slide show". -->

<!ELEMENT slideshow (slide+)>
```

As you can see, the DTD tag starts with `<!` followed by the tag name (ELEMENT). After the tag name comes the name of the element that is being defined (`slideshow`) and, in parentheses, one or more items that indicate the valid contents for that element. In this case, the notation says that a `slideshow` consists of one or more `slide` elements.

Without the plus sign, the definition would be saying that a `slideshow` consists of a single `slide` element. The qualifiers you can add to an element definition are listed in Table 7-2.

Table 7-2 DTD Element Qualifiers

Qualifier	Name	Meaning
?	Question Mark	Optional (zero or one)
*	Asterisk	Zero or more
+	Plus Sign	One or more

You can include multiple elements inside the parentheses in a comma separated list, and use a qualifier on each element to indicate how many instances of that element may occur. The comma-separated list tells which elements are valid and the order they can occur in.

You can also nest parentheses to group multiple items. For an example, after defining an `image` element (coming up shortly), you could declare that every `image` element must be paired with a `title` element in a slide by specifying `((image, title)+)`. Here, the plus sign applies to the `image/title` pair to indicate that one or more pairs of the specified items can occur.

Defining Text and Nested Elements

Now that you have told the parser something about where *not* to expect text, let's see how to tell it where text *can* occur. Add the text highlighted below to define the `slide`, `title`, `item`, and `list` elements:

```
<!ELEMENT slideshow (slide+)>
<!ELEMENT slide (title, item*)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT item (#PCDATA | item)* >
```

The first line you added says that a `slide` consists of a `title` followed by zero or more `item` elements. Nothing new there. The next line says that a `title` consists entirely of *parsed character data* (PCDATA). That's known as "text" in most parts of the country, but in XML-speak it's called "parsed character data". (That distinguishes it from CDATA sections, which contain character data that is not

parsed.) The “#” that precedes PCDATA indicates that what follows is a special word, rather than an element name.

The last line introduces the vertical bar (|), which indicates an *or* condition. In this case, either PCDATA or an `item` can occur. The asterisk at the end says that either one can occur zero or more times in succession. The result of this specification is known as a *mixed-content model*, because any number of `item` elements can be interspersed with the text. Such models must always be defined with `#PCDATA` specified first, some number of alternate items divided by vertical bars (|), and an asterisk (*) at the end.

Save a copy of this DTD as `slideSample1a.dtd`, for use when experimenting with basic DTD processing.

Limitations of DTDs

It would be nice if we could specify that an `item` contains either text, or text followed by one or more list items. But that kind of specification turns out to be hard to achieve in a DTD. For example, you might be tempted to define an `item` like this:

```
<!ELEMENT item (#PCDATA | (#PCDATA, item+)) >
```

That would certainly be accurate, but as soon as the parser sees `#PCDATA` and the vertical bar, it requires the remaining definition to conform to the mixed-content model. This specification doesn't, so you get an error that says: `Illegal mixed content model for 'item'. Found (...`, where the hex character 28 is the angle bracket that ends the definition.

Trying to double-define the `item` element doesn't work, either. A specification like this:

```
<!ELEMENT item (#PCDATA) >
<!ELEMENT item (#PCDATA, item+) >
```

produces a “duplicate definition” warning when the validating parser runs. The second definition is, in fact, ignored. So it seems that defining a mixed content model (which allows `item` elements to be interspersed in text) is about as good as we can do.

In addition to the limitations of the mixed content model mentioned above, there is no way to further qualify the kind of text that can occur where PCDATA has

been specified. Should it contain only numbers? Should be in a date format, or possibly a monetary format? There is no way to say in the context of a DTD.

Finally, note that the DTD offers no sense of hierarchy. The definition for the `title` element applies equally to a `slide` title and to an `item` title. When we expand the DTD to allow HTML-style markup in addition to plain text, it would make sense to restrict the size of an `item` title compared to a `slide` title, for example. But the only way to do that would be to give one of them a different name, such as “`item-title`”. The bottom line is that the lack of hierarchy in the DTD forces you to introduce a “hyphenation hierarchy” (or its equivalent) in your namespace. All of these limitations are fundamental motivations behind the development of schema-specification standards.

Special Element Values in the DTD

Rather than specifying a parenthesized list of elements, the element definition could use one of two special values: `ANY` or `EMPTY`. The `ANY` specification says that the element may contain any other defined element, or `PCDATA`. Such a specification is usually used for the root element of a general-purpose XML document such as you might create with a word processor. Textual elements could occur in any order in such a document, so specifying `ANY` makes sense.

The `EMPTY` specification says that the element contains no contents. So the DTD for e-mail messages that let you “flag” the message with `<flag/>` might have a line like this in the DTD:

```
<!ELEMENT flag EMPTY>
```

Referencing the DTD

In this case, the DTD definition is in a separate file from the XML document. That means you have to reference it from the XML document, which makes the DTD file part of the external subset of the full Document Type Definition (DTD) for the XML file. As you’ll see later on, you can also include parts of the DTD within the document. Such definitions constitute the local subset of the DTD.

Note: The XML written in this section is contained in `slideSample05.xml`. (The browsable version is `slideSample05-xml.html`.)

To reference the DTD file you just created, add the line highlighted below to your `slideSample.xml` file, and save a copy of the file as `slideSample05.xml`:

```
<!-- A SAMPLE set of slides -->

<!DOCTYPE slideshow SYSTEM "slideshow.dtd">

<slideshow
```

Again, the DTD tag starts with “<!”. In this case, the tag name, DOCTYPE, says that the document is a `slideshow`, which means that the document consists of the `slideshow` element and everything within it:

```
<slideshow>
...
</slideshow>
```

This tag defines the `slideshow` element as the root element for the document. An XML document must have exactly one root element. This is where that element is specified. In other words, this tag identifies the document *content* as a `slideshow`.

The DOCTYPE tag occurs after the XML declaration and before the root element. The SYSTEM identifier specifies the location of the DTD file. Since it does not start with a prefix like `http://` or `file:/`, the path is relative to the location of the XML document. Remember the `setDocumentLocator` method? The parser is using that information to find the DTD file, just as your application would to find a file relative to the XML document. A PUBLIC identifier could also be used to specify the DTD file using a unique name—but the parser would have to be able to resolve it

The DOCTYPE specification could also contain DTD definitions within the XML document, rather than referring to an external DTD file. Such definitions would be contained in square brackets, like this:

```
<!DOCTYPE slideshow SYSTEM "slideshow1.dtd" [
    ...local subset definitions here...
]>
```

You’ll take advantage of that facility in a moment to define some entities that can be used in the document.

Documents and Data

Earlier, you learned that one reason you hear about XML *documents*, on the one hand, and XML *data*, on the other, is that XML handles both comfortably, depending on whether text is or is not allowed between elements in the structure.

In the sample file you have been working with, the `slideshow` element is an example of a *data element*—it contains only subelements with no intervening text. The `item` element, on the other hand, might be termed a *document element*, because it is defined to include both text and subelements.

As you work through this tutorial, you will see how to expand the definition of the title element to include HTML-style markup, which will turn it into a document element as well.

Defining Attributes and Entities in the DTD

The DTD you’ve defined so far is fine for use with the nonvalidating parser. It tells where text is expected and where it isn’t, which is all the nonvalidating parser is going to pay attention to. But for use with the validating parser, the DTD needs to specify the valid attributes for the different elements. You’ll do that in this section, after which you’ll define one internal entity and one external entity that you can reference in your XML file.

Defining Attributes in the DTD

Let’s start by defining the attributes for the elements in the slide presentation.

Note: The XML written in this section is contained in `slideshow1b.dtd`. (The browsable version is `slideshow1b-dtd.html`.)

Add the text highlighted below to define the attributes for the `slideshow` element:

```
<!ELEMENT slideshow (slide+)>
<!ATTLIST slideshow
    title      CDATA      #REQUIRED
    date       CDATA      #IMPLIED
    author     CDATA      "unknown"
>
<!ELEMENT slide (title, item*)>
```

The DTD tag `ATTLIST` begins the series of attribute definitions. The name that follows `ATTLIST` specifies the element for which the attributes are being defined. In this case, the element is the `slideshow` element. (Note once again the lack of hierarchy in DTD specifications.)

Each attribute is defined by a series of three space-separated values. Commas and other separators are not allowed, so formatting the definitions as shown above is helpful for readability. The first element in each line is the name of the attribute: `title`, `date`, or `author`, in this case. The second element indicates the type of the data: `CDATA` is character data—unparsed data, once again, in which a left-angle bracket (`<`) will never be construed as part of an XML tag. Table 7–3 presents the valid choices for the attribute type.

Table 7–3 Attribute Types

Attribute Type	Specifies...
(value1 value2 ...)	A list of values separated by vertical bars. (Example below)
CDATA	“Unparsed character data”. (For normal people, a text string.)
ID	A name that no other ID attribute shares.
IDREF	A reference to an ID defined elsewhere in the document.
IDREFS	A space-separated list containing one or more ID references.
ENTITY	The name of an entity defined in the DTD.
ENTITIES	A space-separated list of entities.
NMTOKEN	A valid XML name composed of letters, numbers, hyphens, underscores, and colons.
NMTOKENS	A space-separated list of names.
NOTATION	The name of a DTD-specified notation, which describes a non-XML data format, such as those used for image files.*

*This is a rapidly obsolescing specification which will be discussed in greater length towards the end of this section.

When the attribute type consists of a parenthesized list of choices separated by vertical bars, the attribute must use one of the specified values. For an example, add the text highlighted below to the DTD:

```
<!ELEMENT slide (title, item*)>
<!ATTLIST slide
    type    (tech | exec | all) #IMPLIED
>
<!ELEMENT title (#PCDATA)>
<!ELEMENT item (#PCDATA | item)* >
```

This specification says that the `slide` element's `type` attribute must be given as `type="tech"`, `type="exec"`, or `type="all"`. No other values are acceptable. (DTD-aware XML editors can use such specifications to present a pop-up list of choices.)

The last entry in the attribute specification determines the attribute's default value, if any, and tells whether or not the attribute is required. Table 7-4 shows the possible choices.

Table 7-4 Attribute-Specification Parameters

Specification	Specifies...
#REQUIRED	The attribute value must be specified in the document.
#IMPLIED	The value need not be specified in the document. If it isn't, the application will have a default value it uses.
"defaultValue"	The default value to use, if a value is not specified in the document.
#FIXED "fixedValue"	The value to use. If the document specifies any value at all, it must be the same.

Finally, save a copy of the DTD as `slideshow1b.dtd`, for use when experimenting with attribute definitions.

Defining Entities in the DTD

So far, you’ve seen predefined entities like `&`; and you’ve seen that an attribute can reference an entity. It’s time now for you to learn how to define entities of your own.

Note: The XML you’ll create here is contained in `slideSample06.xml`. (The browsable version is `slideSample06-xml.html`.)

Add the text highlighted below to the DOCTYPE tag in your XML file:

```
<!DOCTYPE slideshow SYSTEM "slideshow.dtd" [  
  <!ENTITY product "WonderWidget">  
  <!ENTITY products "WonderWidgets">  
>
```

The ENTITY tag name says that you are defining an entity. Next comes the name of the entity and its definition. In this case, you are defining an entity named “product” that will take the place of the product name. Later when the product name changes (as it most certainly will), you will only have to change the name one place, and all your slides will reflect the new value.

The last part is the substitution string that replaces the entity name whenever it is referenced in the XML document. The substitution string is defined in quotes, which are not included when the text is inserted into the document.

Just for good measure, we defined two versions, one singular and one plural, so that when the marketing mavens come up with “Wally” for a product name, you will be prepared to enter the plural as “Wallies” and have it substituted correctly.

Note: Truth be told, this is the kind of thing that really belongs in an external DTD. That way, all your documents can reference the new name when it changes. But, hey, this is an example...

Now that you have the entities defined, the next step is to reference them in the slide show. Make the changes highlighted below to do that:

```
<slideshow
  title="WonderWidget&product; Slide Show"
  ...

  <!-- TITLE SLIDE -->
  <slide type="all">
    <title>Wake up to WonderWidgets&products;!</title>
  </slide>

  <!-- OVERVIEW -->
  <slide type="all">
    <title>Overview</title>
    <item>Why <em>WonderWidgets&products;</em> are
great</item>
    <item/>
    <item>Who <em>buys</em> WonderWidgets&products;</item>
  </slide>
```

The points to notice here are that entities you define are referenced with the same syntax (&entityName;) that you use for predefined entities, and that the entity can be referenced in an attribute value as well as in an element's contents.

When you echo this version of the file with an XML parser, here is the kind of thing you'll see:

```
Wake up to WonderWidgets!
```

Note that the product name has been substituted for the entity reference.

To finish, save a copy of the file as `slideSample06.xml`.

Additional Useful Entities

Here are several other examples for entity definitions that you might find useful when you write an XML document:

```
<!ENTITY ldquo "&#147;"> <!-- Left Double Quote -->
<!ENTITY rdquo "&#148;"> <!-- Right Double Quote -->
<!ENTITY trade "&#153;"> <!-- Trademark Symbol (TM) -->
<!ENTITY rtrade "&#174;"> <!-- Registered Trademark (R) -->
<!ENTITY copyr "&#169;"> <!-- Copyright Symbol -->
```


Referencing External Entities

You can also use the SYSTEM or PUBLIC identifier to name an entity that is defined in an external file. You'll do that now.

Note: The XML defined here is contained in `slideSample07.xml` and in `copyright.xml`. (The browsable versions are `slideSample07-xml.html` and `copyright-xml.html`.)

To reference an external entity, add the text highlighted below to the DOCTYPE statement in your XML file:

```
<!DOCTYPE slideshow SYSTEM "slideshow.dtd" [
  <!ENTITY product "WonderWidget">
  <!ENTITY products "WonderWidgets">
  <!ENTITY copyright SYSTEM "copyright.xml">
]>
```

This definition references a copyright message contained in a file named `copyright.xml`. Create that file and put some interesting text in it, perhaps something like this:

```
<!-- A SAMPLE copyright -->

This is the standard copyright message that our lawyers
make us put everywhere so we don't have to shell out a
million bucks every time someone spills hot coffee in their
lap...
```

Finally, add the text highlighted below to your `slideSample.xml` file to reference the external entity, and save a copy of the file as `slideSample07.html`:

```
<!-- TITLE SLIDE -->
...
</slide>

<!-- COPYRIGHT SLIDE -->
<slide type="all">
  <item>&copyright;</item>
</slide>
```

You could also use an external entity declaration to access a servlet that produces the current date using a definition something like this:

```
<!ENTITY currentDate SYSTEM
    "http://www.example.com/servlet/CurrentDate?fmt=dd-MMM-
    yyyy">
```

You would then reference that entity the same as any other entity:

```
Today's date is &currentDate;.
```

When you echo the latest version of the slide presentation with an XML parser, here is what you'll see:

```
...
<slide type="all">
  <item>
    This is the standard copyright message that our lawyers
    make us put everywhere so we don't have to shell out a
    million bucks every time someone spills hot coffee in their
    lap...
  </item>
</slide>
...
```

You'll notice that the newline which follows the comment in the file is echoed as a character, but that the comment itself is ignored. That is the reason that the copyright message appears to start on the next line after the `<item>` element, instead of on the same line—the first character echoed is actually the newline that follows the comment.

Summarizing Entities

An entity that is referenced in the document content, whether internal or external, is termed a general entity. An entity that contains DTD specifications that are referenced from within the DTD is termed a parameter entity. (More on that later.)

An entity which contains XML (text and markup), and which is therefore parsed, is known as a *parsed entity*. An entity which contains binary data (like images) is known as an *unparsed entity*. (By its very nature, it must be external.) We'll be discussing references to unparsed entities in the next section of this tutorial.

Referencing Binary Entities

This section discusses the options for referencing binary files like image files and multimedia data files.

Using a MIME Data Type

There are two ways to go about referencing an unparsed entity like a binary image file. One is to use the DTD's NOTATION-specification mechanism. However, that mechanism is a complex, non-intuitive holdover that mostly exists for compatibility with SGML documents. We will have occasion to discuss it in a bit more depth when we look at the DTDHandler API, but suffice it for now to say that the combination of the recently defined XML namespaces standard, in conjunction with the MIME data types defined for electronic messaging attachments, together provide a much more useful, understandable, and extensible mechanism for referencing unparsed external entities.

Note: The XML described here is in `slideshow1b.dtd`. It shows how binary references can be made, assuming that the application which will be processing the XML data knows how to handle such references.

To set up the slideshow to use image files, add the text highlighted below to your `slideshow1b.dtd` file:

```
<!ELEMENT slide (image?, title, item*)>
<!ATTLIST slide
    type    (tech | exec | all) #IMPLIED
>
<!ELEMENT title (#PCDATA)>
<!ELEMENT item (#PCDATA | item)* >
<!ELEMENT image EMPTY>
<!ATTLIST image
    alt      CDATA      #IMPLIED
    src      CDATA      #REQUIRED
    type     CDATA      "image/gif"
>
```

These modifications declare `image` as an optional element in a `slide`, define it as empty element, and define the attributes it requires. The `image` tag is patterned after the HTML 4.0 tag, `img`, with the addition of an image-type specifier, `type`. (The `img` tag is defined in the HTML 4.0 Specification.)

The `image` tag's attributes are defined by the `ATTLIST` entry. The `alt` attribute, which defines alternate text to display in case the image can't be found, accepts character data (CDATA). It has an "implied" value, which means that it is optional, and that the program processing the data knows enough to substitute something like "Image not found". On the other hand, the `src` attribute, which names the image to display, is required.

The `type` attribute is intended for the specification of a MIME data type, as defined at <ftp://ftp.isi.edu/in-notes/iana/assignments/media-types/>. It has a default value: `image/gif`.

Note: It is understood here that the character data (CDATA) used for the `type` attribute will be one of the MIME data types. The two most common formats are: `image/gif`, and `image/jpeg`. Given that fact, it might be nice to specify an attribute list here, using something like:

```
type ("image/gif", "image/jpeg")
```

That won't work, however, because attribute lists are restricted to name tokens. The forward slash isn't part of the valid set of name-token characters, so this declaration fails. Besides that, creating an attribute list in the DTD would limit the valid MIME types to those defined today. Leaving it as CDATA leaves things more open ended, so that the declaration will continue to be valid as additional types are defined.

In the document, a reference to an image named "intro-pic" might look something like this:

```
<image src="image/intro-pic.gif", alt="Intro Pic",  
type="image/gif" />
```

The Alternative: Using Entity References

Using a MIME data type as an attribute of an element is a mechanism that is flexible and expandable. To create an external ENTITY reference using the notation mechanism, you need DTD NOTATION elements for JPEG and GIF data. Those can of course be obtained from some central repository. But then you need to define a different ENTITY element for each image you intend to reference! In other words, adding a new image to your document always requires both a new entity definition in the DTD and a reference to it in the document. Given the anticipated ubiquity of the HTML 4.0 specification, the newer standard is to use

the MIME data types and a declaration like `image`, which assumes the application knows how to process such elements.

Defining Parameter Entities and Conditional Sections

Just as a general entity lets you reuse XML data in multiple places, a parameter entity lets you reuse parts of a DTD in multiple places. In this section of the tutorial, you'll see how to define and use parameter entities. You'll also see how to use parameter entities with conditional sections in a DTD.

Creating and Referencing a Parameter Entity

Recall that the existing version of the slide presentation could not be validated because the document used `` tags, and those are not part of the DTD. In general, we'd like to use a whole variety of HTML-style tags in the text of a slide, not just one or two, so it makes more sense to use an existing DTD for XHTML than it does to define all the tags we might ever need. A parameter entity is intended for exactly that kind of purpose.

Note: The DTD specifications shown here are contained in `slideshow2.dtd` and `xhtml.dtd`. The XML file that references it is `slideSample08.xml`. (The browsable versions are `slideshow2-dtd.html` and `slideSample08-xml.html`.)

Open your DTD file for the slide presentation and add the text highlighted below to define a parameter entity that references an external DTD file:

```
<!ELEMENT slide (image?, title?, item*)>
<!ATTLIST slide
    ...
>

<!ENTITY % xhtml SYSTEM "xhtml.dtd">
%xhtml;

<!ELEMENT title ...
```

Here, you used an `<!ENTITY>` tag to define a parameter entity, just as for a general entity, but using a somewhat different syntax. You included a percent sign

(%) before the entity name when you defined the entity, and you used the percent sign instead of an ampersand when you referenced it.

Also, note that there are always two steps for using a parameter entity. The first is to define the entity name. The second is to reference the entity name, which actually does the work of including the external definitions in the current DTD. Since the URI for an external entity could contain slashes (/) or other characters that are not valid in an XML name, the definition step allows a valid XML name to be associated with an actual document. (This same technique is used in the definition of namespaces, and anywhere else that XML constructs need to reference external documents.)

Notes:

- The DTD file referenced by this definition is `xhtml.dtd`. You can either copy that file to your system or modify the `SYSTEM` identifier in the `<!ENTITY>` tag to point to the correct URL.
- This file is a small subset of the XHTML specification, loosely modeled after the Modularized XHTML draft, which aims at breaking up the DTD for XHTML into bite-sized chunks, which can then be combined to create different XHTML subsets for different purposes. When work on the modularized XHTML draft has been completed, this version of the DTD should be replaced with something better. For now, this version will suffice for our purposes.

The whole point of using an XHTML-based DTD was to gain access to an entity it defines that covers HTML-style tags like `` and ``. Looking through `xhtml.dtd` reveals the following entity, which does exactly what we want:

```
<!ENTITY % inline "#PCDATA|em|b|a|img|br">
```

This entity is a simpler version of those defined in the Modularized XHTML draft. It defines the HTML-style tags we are most likely to want to use -- emphasis, bold, and break, plus a couple of others for images and anchors that we may or may not use in a slide presentation. To use the `inline` entity, make the changes highlighted below in your DTD file:

```
<!ELEMENT title (#PCDATA %inline;)*>
<!ELEMENT item (#PCDATA %inline; | item)* >
```

These changes replaced the simple `#PCDATA` item with the `inline` entity. It is important to notice that `#PCDATA` is first in the `inline` entity, and that `inline` is first wherever we use it. That is required by XML's definition of a mixed-content

model. To be in accord with that model, you also had to add an asterisk at the end of the `title` definition.

Save the DTD as `slideshow2.dtd`, for use when experimenting with parameter entities.

Note: The Modularized XHTML DTD defines both `inline` and `Inline` entities, and does so somewhat differently. Rather than specifying `#PCDATA|em|b|a|img|Br`, their definitions are more like `(#PCDATA|em|b|a|img|Br)*`. Using one of those definitions, therefore, looks more like this:

```
<!ELEMENT title %Inline; >
```

Conditional Sections

Before we proceed with the next programming exercise, it is worth mentioning the use of parameter entities to control *conditional sections*. Although you cannot conditionalize the content of an XML document, you can define conditional sections in a DTD that become part of the DTD only if you specify `include`. If you specify `ignore`, on the other hand, then the conditional section is not included.

Suppose, for example, that you wanted to use slightly different versions of a DTD, depending on whether you were treating the document as an XML document or as a SGML document. You could do that with DTD definitions like the following:

```
someExternal.dtd:
<![ INCLUDE [
    ... XML-only definitions
]]>
<![ IGNORE [
    ... SGML-only definitions
]]>
... common definitions
```

The conditional sections are introduced by “<![”, followed by the `INCLUDE` or `IGNORE` keyword and another “[”. After that comes the contents of the conditional section, followed by the terminator: “]]>”. In this case, the XML definitions are included, and the SGML definitions are excluded. That’s fine for XML documents, but you can’t use the DTD for SGML documents. You could change the keywords, of course, but that only reverses the problem.

The solution is to use references to parameter entities in place of the `INCLUDE` and `IGNORE` keywords:

```
someExternal.dtd:
<![ %XML; [
    ... XML-only definitions
]]>
<![ %SGML; [
    ... SGML-only definitions
]]>
... common definitions
```

Then each document that uses the DTD can set up the appropriate entity definitions:

```
<!DOCTYPE foo SYSTEM "someExternal.dtd" [
  <!ENTITY % XML "INCLUDE" >
  <!ENTITY % SGML "IGNORE" >
]>
<foo>
...
</foo>
```

This procedure puts each document in control of the DTD. It also replaces the `INCLUDE` and `IGNORE` keywords with variable names that more accurately reflect the purpose of the conditional section, producing a more readable, self-documenting version of the DTD.

Resolving A Naming Conflict

The XML structures you have created thus far have actually encountered a small naming conflict. It seems that `xhtml.dtd` defines a `title` element which is entirely different from the `title` element defined in the `slideshow DTD`. Because there is no hierarchy in the DTD, these two definitions conflict.

Note: The Modularized XHTML DTD also defines a `title` element that is intended to be the document title, so we can't avoid the conflict by changing `xhtml.dtd`—the problem would only come back to haunt us later.

You could use XML namespaces to resolve the conflict. You'll take a look at that approach in the next section. Alternatively, you could use one of the more hierarchical schema proposals described in *Schema Standards* (page 192). The sim-

plest way to solve the problem for now, though, is simply to rename the title element in `slideshow.dtd`.

Note: The XML shown here is contained in `slideshow3.dtd` and `slideSample09.xml`, which references `copyright.xml` and `xhtml.dtd`. (The browsable versions are `slideshow3-dtd.html`, `slideSample09-xml.html`, `copyright-xml.html`, and `xhtml-dtd.html`.)

To keep the two title elements separate, you'll create a "hyphenation hierarchy". Make the changes highlighted below to change the name of the title element in `slideshow.dtd` to `slide-title`:

```
<!ELEMENT slide (image?, slide-title?, item*)>
<!ATTLIST slide
    type    (tech | exec | all) #IMPLIED
>

<!-- Defines the %inline; declaration -->
<!ENTITY % xhtml SYSTEM "xhtml.dtd">
%xhtml;

<!ELEMENT slide-title (%inline;)*>
```

Save this DTD as `slideshow3.dtd`.

The next step is to modify the XML file to use the new element name. To do that, make the changes highlighted below:

```
...
<slide type="all">
<slide-title>Wake up to ... </slide-title>
</slide>

...

<!-- OVERVIEW -->
<slide type="all">
<slide-title>Overview</slide-title>
<item>...
```

Save a copy of this file as `slideSample09.xml`.

Using Namespaces

As you saw earlier, one way or another it is necessary to resolve the conflict between the `title` element defined in `slideshow.dtd` and the one defined in `xhtml.dtd` when the same name is used for different purposes. In the previous exercise, you hyphenated the name in order to put it into a different “namespace”. In this section, you’ll see how to use the XML namespace standard to do the same thing without renaming the element.

The primary goal of the namespace specification is to let the document author tell the parser which DTD or schema to use when parsing a given element. The parser can then consult the appropriate DTD or schema for an element definition. Of course, it is also important to keep the parser from aborting when a “duplicate” definition is found, and yet still generate an error if the document references an element like `title` without *qualifying* it (identifying the DTD or schema to use for the definition).

Note: Namespaces apply to attributes as well as to elements. In this section, we consider only elements. For more information on attributes, consult the namespace specification at <http://www.w3.org/TR/REC-xml-names/>.

Defining a Namespace in a DTD

In a DTD, you define a namespace that an element belongs to by adding an attribute to the element’s definition, where the attribute name is `xmlns` (“xml namespace”). For example, you could do that in `slideshow.dtd` by adding an entry like the following in the `title` element’s attribute-list definition:

```
<!ELEMENT title (%inline;)*>
<!ATTLIST title
  xmlns CDATA #FIXED "http://www.example.com/slideshow"
>
```

Declaring the attribute as `FIXED` has several important features:

- It prevents the document from specifying any non-matching value for the `xmlns` attribute.
- The element defined in this DTD is made unique (because the parser understands the `xmlns` attribute), so it does not conflict with an element

that has the same name in another DTD. That allows multiple DTDs to use the same element name without generating a parser error.

- When a document specifies the `xmlns` attribute for a tag, the document selects the element definition with a matching attribute.

To be thorough, every element name in your DTD would get the exact same attribute, with the same value. (Here, though, we're only concerned about the `title` element.) Note, too, that you are using a CDATA string to supply the URI. In this case, we've specified an URL. But you could also specify a URN, possibly by specifying a prefix like `urn:` instead of `http:`. (URNs are currently being researched. They're not seeing a lot of action at the moment, but that could change in the future.)

Referencing a Namespace

When a document uses an element name that exists in only one of the DTDs or schemas it references, the name does not need to be qualified. But when an element name that has multiple definitions is used, some sort of qualification is a necessity.

Note: In point of fact, an element name is always qualified by its *default namespace*, as defined by name of the DTD file it resides in. As long as there is only one definition for the name, the qualification is implicit.

You qualify a reference to an element name by specifying the `xmlns` attribute, as shown here:

```
<title xmlns="http://www.example.com/slideshow">
  Overview
</title>
```

The specified namespace applies to that element, and to any elements contained within it.

Defining a Namespace Prefix

When you only need one namespace reference, it's not such a big deal. But when you need to make the same reference several times, adding `xmlns` attributes becomes unwieldy. It also makes it harder to change the name of the namespace at a later date.

The alternative is to define a *namespace prefix*, which is as simple as specifying `xmlns`, a colon (:), and the prefix name before the attribute value, as shown here:

```
<SL:slideshow xmlns:SL='http://www.example.com/slideshow'
...>
...
</SL:slideshow>
```

This definition sets up `SL` as a prefix that can be used to qualify the current element name and any element within it. Since the prefix can be used on any of the contained elements, it makes the most sense to define it on the XML document's root element, as shown here.

Note: The namespace URI can contain characters which are not valid in an XML name, so it cannot be used as a prefix directly. The prefix definition associates an XML name with the URI, which allows the prefix name to be used instead. It also makes it easier to change references to the URI in the future.

When the prefix is used to qualify an element name, the end-tag also includes the prefix, as highlighted here:

```
<SL:slideshow xmlns:SL='http://www.example.com/slideshow'
...>
...
<slide>
  <SL:title>Overview</SL:title>
</slide>
...
</SL:slideshow>
```

Finally, note that multiple prefixes can be defined in the same element, as shown here:

```
<SL:slideshow xmlns:SL='http://www.example.com/slideshow'
  xmlns:xhtml='urn:...'>
...
</SL:slideshow>
```

With this kind of arrangement, all of the prefix definitions are together in one place, and you can use them anywhere they are needed in the document. This example also suggests the use of URN to define the `xhtml` prefix, instead of an URL. That definition would conceivably allow the application to reference a

local copy of the XHTML DTD or some mirrored version, with a potentially beneficial impact on performance.

Designing an XML Data Structure

This section covers some heuristics you can use when making XML design decisions.

Saving Yourself Some Work

Whenever possible, use an existing schema definition. It's usually a lot easier to ignore the things you don't need than to design your own from scratch. In addition, using a standard DTD makes data interchange possible, and may make it possible to use data-aware tools developed by others.

So, if an industry standard exists, consider referencing that DTD with an external parameter entity. One place to look for industry-standard DTDs is at the repository created by the Organization for the Advancement of Structured Information Standards (OASIS) at <http://www.XML.org>. Another place to check is CommerceOne's XML Exchange at <http://www.xmlx.com>, which is described as "a repository for creating and sharing document type definitions".

Note: Many more good thoughts on the design of XML structures are at the OASIS page, <http://www.oasis-open.org/cover/elementsAndAttrs.html>.

Attributes and Elements

One of the issues you will encounter frequently when designing an XML structure is whether to model a given data item as a subelement or as an attribute of an existing element. For example, you could model the title of a slide either as:

```
<slide>
  <title>This is the title</title>
</slide>
```

or as:

```
<slide title="This is the title">...</slide>
```

In some cases, the different characteristics of attributes and elements make it easy to choose. Let's consider those cases first, and then move on to the cases where the choice is more ambiguous.

Forced Choices

Sometimes, the choice between an attribute and an element is forced on you by the nature of attributes and elements. Let's look at a few of those considerations:

The data contains substructures

In this case, the data item must be modeled as an *element*. It can't be modeled as an attribute, because attributes take only simple strings. So if the title can contain emphasized text like this: The `Best` Choice, then the title must be an element.

The data contains multiple lines

Here, it also makes sense to use an *element*. Attributes need to be simple, short strings or else they become unreadable, if not unusable.

Multiple occurrences are possible

Whenever an item can occur multiple times, like paragraphs in an article, it must be modeled as an *element*. The element that contains it can only have one attribute of a particular kind, but it can have many subelements of the same type.

The data changes frequently

When the data will be frequently modified with an editor, it may make sense to model it as an *element*. Many XML-aware editors make it easy modify element data, while attributes can be somewhat harder to get to.

The data is a small, simple string that rarely if ever changes

This is data that can be modeled as an *attribute*. However, just because you *can* does not mean that you should. Check the "Stylistic Choices" section next, to be sure.

Using DTDs when the data is confined to a small number of fixed choices

Here is one time when it really makes sense to use an *attribute*. A DTD can prevent an attribute from taking on any value that is not in the preapproved list, but it cannot similarly restrict an element. (With a schema on the other hand, both attributes and elements can be restricted.)

Stylistic Choices

As often as not, the choices are not as cut and dried as those shown above. When the choice is not forced, you need a sense of “style” to guide your thinking. The question to answer, then, is what makes good XML style, and why.

Defining a sense of style for XML is, unfortunately, as nebulous a business as defining “style” when it comes to art or music. There are a few ways to approach it, however. The goal of this section is to give you some useful thoughts on the subject of “XML style”.

Visibility

One heuristic for thinking about XML elements and attributes uses the concept of *visibility*. If the data is intended to be shown—to be displayed to some end user—then it should be modeled as an element. On the other hand, if the information guides XML processing but is never seen by a user, then it may be better to model it as an attribute. For example, in order-entry data for shoes, shoe size would definitely be an element. On the other hand, a manufacturer’s code number would be reasonably modeled as an attribute.

Consumer / Provider

Another way of thinking about the visibility heuristic is to ask who is the consumer and/or provider of the information. The shoe size is entered by a human sales clerk, so it’s an element. The manufacturer’s code number for a given shoe model, on the other hand, may be wired into the application or stored in a database, so that would be an attribute. (If it were entered by the clerk, though, it should perhaps be an element.)

Container vs. Contents

Perhaps the best way of thinking about elements and attributes is to think of an element as a *container*. To reason by analogy, the *contents* of the container (water or milk) correspond to XML data modeled as elements. Such data is essentially variable. On the other hand, *characteristics* of the container (blue or white pitcher) can be modeled as attributes. That kind of information tends to be more immutable. Good XML style will, in some consistent way, separate each container’s contents from its characteristics.

To show these heuristics at work: In a slideshow the type of the slide (executive or technical) is best modeled as an attribute. It is a characteristic of the slide that lets it be selected or rejected for a particular audience. The title of the slide, on the other hand, is part of its contents. The visibility heuristic is also satisfied here. When the slide is displayed, the title is shown but the type of the slide isn’t. Finally, in this example, the consumer of the title information is the presentation

audience, while the consumer of the type information is the presentation program.

Normalizing Data

In *Saving Yourself Some Work* (page 233), you saw that it is a good idea to define an external entity that you can reference in an XML document. Such an entity has all the advantages of a modularized routine—changing that one copy affects every document that references it. The process of eliminating redundancies is known as *normalizing*, so defining entities is one good way to normalize your data.

In an HTML file, the only way to achieve that kind of modularity is with HTML links—but of course the document is then fragmented, rather than whole. XML entities, on the other hand, suffer no such fragmentation. The entity reference acts like a macro—the entity’s contents are expanded in place, producing a whole document, rather than a fragmented one. And when the entity is defined in an external file, multiple documents can reference it.

The considerations for defining an entity reference, then, are pretty much the same as those you would apply to modularized program code:

- Whenever you find yourself writing the same thing more than once, think entity. That lets you write it one place and reference it multiple places.
- If the information is likely to change, especially if it is used in more than one place, definitely think in terms of defining an entity. An example is defining `productName` as an entity so that you can easily change the documents when the product name changes.
- If the entity will never be referenced anywhere except in the current file, define it in the `local_subset` of the document’s DTD, much as you would define a method or inner class in a program.
- If the entity will be referenced from multiple documents, define it as an external entity, the same way that would define any generally usable class as an external class.

External entities produce modular XML that is smaller, easier to update and maintain. They can also make the resulting document somewhat more difficult to visualize, much as a good OO design can be easy to change, once you understand it, but harder to wrap your head around at first.

You can also go overboard with entities. At an extreme, you could make an entity reference for the word “the”—it wouldn’t buy you much, but you could do it.

Note: The larger an entity is, the less likely it is that changing it will have unintended effects. When you define an external entity that covers a whole section on installation instructions, for example, making changes to the section is unlikely to make any of the documents that depend on it come out wrong. Small inline substitutions can be more problematic, though. For example, if `productName` is defined as an entity, the name change can be to a different part of speech, and that can produce! Suppose the product name is something like “HtmlEdit”. That’s a verb. So you write a sentence that becomes, “You can HtmlEdit your file...” after the entity-substitution occurs. That sentence reads fine, because the verb fits well in that context. But if the name is eventually changed to “HtmlEditor”, the sentence becomes “You can HtmlEditor your file...”, which clearly doesn’t work. Still, even if such simple substitutions can sometimes get you in trouble, they can potentially save a lot of time. (One alternative would be to set up entities named `productNoun`, `productVerb`, `productAdj`, and `productAdverb`!)

Normalizing DTDs

Just as you can normalize your XML document, you can also normalize your DTD declarations by factoring out common pieces and referencing them with a parameter entity. Factoring out the DTDs (also known as modularizing or normalizing) gives the same advantages and disadvantages as normalized XML—easier to change, somewhat more difficult to follow.

You can also set up conditionalized DTDs. If the number and size of the conditional sections is small relative to the size of the DTD as a whole, that can let you “single source” a DTD that you can use for multiple purposes. If the number of conditional sections gets large, though, the result can be a complex document that is difficult to edit.

Summary

Congratulations! You have now created a number of XML files that you can use for testing purposes. Here's a table that describes the files you have constructed.

Table 7-5 Listing of Sample XML Files

File	Contents
slideSample01.xml	A basic file containing a few elements and attributes, as well as comments.
slideSample02.xml	Includes a processing instruction.
SlideSampleBad1.xml	A file that is <i>not</i> well-formed.
slideSample03.xml	Includes a simple entity reference (<).
slideSample04.xml	Contains a CDATA section.
slideSample05.xml	References either a simple external DTD for elements (slideshow1a.dtd), for use with a nonvalidating parser, or else a DTD that defines attributes (slideshow1b.dtd) for use with a validating parser.
slideSample06.xml	Defines two entities locally (product and products), and references slideshow1b.dtd.
slideSample07.xml	References an external entity defined locally (copyright.xml), and references slideshow1b.dtd.
slideSample08.xml	References xhtml.dtd using a parameter entity in slideshow2.dtd, producing a naming conflict, since title is declared in both.
slideSample09.xml	Changes the title element to slide-title, so it can reference xhtml.dtd using a parameter entity in slideshow3.dtd without conflict.

Introduction to Web Services

Maydene Fisher

WEB services, in the general meaning of the term, are services offered via the Web. In a typical Web services scenario, a business application sends a request to a service at a given URL using the SOAP protocol over HTTP. The service receives the request, processes it, and returns a response. An often-cited example of a Web service is that of a stock quote service, in which the request asks for the current price of a specified stock, and the response gives the stock price. This is one of the simplest forms of a Web service in that the request is filled almost immediately, with the request and response being parts of the same method call.

Another example could be a service that maps out an efficient route for the delivery of goods. In this case, a business sends a request containing the delivery destinations, which the service processes to determine the most cost-effective delivery route. The time it takes to return the response depends on the complexity of the routing, so the response will probably be sent as an operation that is separate from the request.

Web services and consumers of Web services are typically businesses, making Web services predominantly business-to-business (B-to-B) transactions. An enterprise can be the provider of Web services and also the consumer of other Web services. For example, a wholesale distributor of spices could be in the consumer role when it uses a Web service to check on the availability of vanilla beans and in the provider role when it supplies prospective customers with different vendors' prices for vanilla beans.

The Role of XML and the Java Platform

Web services depend on the ability of parties to communicate with each other even if they are using different information systems. Extensible Markup Language (XML), a markup language that makes data portable, is a key technology in addressing this need. Enterprises have discovered the benefits of using XML for the integration of data both internally for sharing legacy data among departments and externally for sharing data with other enterprises. As a result, XML is increasingly being used for enterprise integration applications, both in tightly coupled and loosely coupled systems. Because of this data integration ability, XML has become the underpinning for Web-related computing.

Web services also depend on the ability of enterprises using different computing platforms to communicate with each other. This requirement makes the Java platform, which makes code portable, the natural choice for developing Web services. This choice is even more attractive as the new Java APIs for XML become available, making it easier and easier to use XML from the Java programming language. These APIs are summarized later in this introduction and explained in detail in the tutorials for each API.

In addition to data portability and code portability, Web services need to be scalable, secure, and efficient, especially as they grow. The Java 2 Platform, Enterprise Edition is specifically designed to fill just such needs. It facilitates the really hard part of developing Web services, which is programming the infrastructure, or “plumbing.” This infrastructure includes features such as security, distributed transaction management, and connection pool management, all of which are essential for industrial strength Web services. And because components are reusable, development time is substantially reduced.

Because XML and the Java platform work so well together, they have come to play a central role in Web services. In fact, the advantages offered by the Java APIs for XML and the J2EE platform make them the ideal combination for deploying Web services.

The APIs described in this chapter complement and layer on top of the J2EE APIs. These APIs enable the Java community, developers, and tool and container vendors to start developing Web services applications and products using standard Java APIs that maintain the fundamental Write Once, Run Anywhere™ proposition of Java technology.

In the Sun ONE Application Server, these APIs are found in the `<S1AS7_HOME>/share/lib` directory and are automatically loaded in the server's

classpath. In the Sun ONE Studio, these APIs are available in the directory `<S1STUDIO_HOME>/jwsdp/common/lib`.

The remainder of this introduction first gives an overview of the Java APIs for XML, explaining what they do and how they make writing Web applications easier. It then describes each of the APIs individually and then presents a scenario that illustrates how they can work together.

The tutorials that follow give more detailed explanations and walk you through how to use the Java APIs for XML to build applications for Web services. They also provide sample applications that you can run.

Overview of the Java APIs for XML

The Java APIs for XML let you write your Web applications entirely in the Java programming language. They fall into two broad categories: those that deal directly with processing XML documents and those that deal with procedures.

- Document-oriented
 - Java API for XML Processing (JAXP) — processes XML documents using various parsers
- Procedure-oriented
 - Java API for XML-based RPC (JAX-RPC) — sends SOAP method calls to remote parties over the Internet and receives the results
 - Java API for XML Messaging (JAXM) — sends SOAP messages over the Internet in a standard way
 - Java API for XML Registries (JAXR) — provides a standard way to access business registries and share information

Perhaps the most important feature of the Java APIs for XML is that they all support industry standards, thus ensuring interoperability. Various network interoperability standards groups, such as the World Wide Web Consortium (W3C) and the Organization for the Advancement of Structured Information Standards (OASIS), have been defining standard ways of doing things so that businesses who follow these standards can make their data and applications work together.

Another feature of the Java APIs for XML is that they allow a great deal of flexibility. Users have flexibility in how they use the APIs. For example, JAXP code can use various tools for processing an XML document, and JAXM code can use various messaging protocols on top of SOAP. Implementers have flexibility as well. The Java APIs for XML define strict compatibility requirements to ensure

that all implementations deliver the standard functionality, but they also give developers a great deal of freedom to provide implementations tailored to specific uses.

The following sections discuss each of these APIs, giving an overview and a feel for how to use them.

JAXP

The Java API for XML Processing (JAXP) makes it easy to process XML data using applications written in the Java programming language. JAXP leverages the parser standards SAX (Simple API for XML Parsing) and DOM (Document Object Model) so that you can choose to parse your data as a stream of events or to build a tree-structured representation of it. The latest versions of JAXP also support the XSLT (XML Stylesheet Language Transformations) standard, giving you control over the presentation of the data and enabling you to convert the data to other XML documents or to other formats, such as HTML. JAXP also provides namespace support, allowing you to work with schemas that might otherwise have naming conflicts.

Designed to be flexible, JAXP allows you to use any XML-compliant parser from within your application. It does this with what is called a pluggability layer, which allows you to plug in an implementation of the SAX or DOM APIs. The pluggability layer also allows you to plug in an XSL processor, which lets you transform your XML data in a variety of ways, including the way it is displayed.

The SAX API

The Simple API for XML (SAX) defines an API for an event-based parser. Being event-based means that the parser reads an XML document from beginning to end, and each time it recognizes a syntax construction, it notifies the application that is running it. The SAX parser notifies the application by calling methods from the `ContentHandler` interface. For example, when the parser comes to a less than symbol (“<”), it calls the `startElement` method; when it comes to character data, it calls the `characters` method; when it comes to the less than symbol followed by a slash (“</”), it calls the `endElement` method, and so on. To illustrate, let’s look at part of the example XML document from the

first section and walk through what the parser does for each line. (For simplicity, calls to the method `ignoreableWhiteSpace` are not included.)

```
<priceList>    [parser calls startElement]
  <coffee>      [parser calls startElement]
    <name>Mocha Java</name>    [parser calls startElement,
                              characters, and endElement]
    <price>11.95</price>      [parser calls startElement,
                              characters, and endElement]
  </coffee>      [parser calls endElement]
```

The default implementations of the methods that the parser calls do nothing, so you need to write a subclass implementing the appropriate methods to get the functionality you want. For example, suppose you want to get the price per pound for Mocha Java. You would write a class extending `DefaultHandler` (the default implementation of `ContentHandler`) in which you write your own implementations of the methods `startElement` and `characters`.

You first need to create a `SAXParser` object from a `SAXParserFactory` object. You would call the method `parse` on it, passing it the price list and an instance of your new handler class (with its new implementations of the methods `startElement` and `characters`). In this example, the price list is a file, but the `parse` method can also take a variety of other input sources, including an `InputStream` object, a URL, and an `InputSource` object.

```
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser saxParser = factory.newSAXParser();
saxParser.parse("priceList.xml", handler);
```

The result of calling the method `parse` depends, of course, on how the methods in `handler` were implemented. The SAX parser will go through the file `priceList.xml` line by line, calling the appropriate methods. In addition to the methods already mentioned, the parser will call other methods such as `startDocument`, `endDocument`, `ignoreableWhiteSpace`, and `processingInstructions`, but these methods still have their default implementations and thus do nothing.

The following method definitions show one way to implement the methods `characters` and `startElement` so that they find the price for Mocha Java and print it out. Because of the way the SAX parser works, these two methods work together to look for the name element, the characters “Mocha Java”, and the price element immediately following Mocha Java. These methods use three flags to keep track of which conditions have been met. Note that the SAX parser

will have to invoke both methods more than once before the conditions for printing the price are met.

```

public void startElement(..., String elementName, ...){
    if(elementName.equals("name")){
        inName = true;
    } else if(elementName.equals("price") && inMochaJava ){
        inPrice = true;
        inName = false;
    }
}

public void characters(char [] buf, int offset, int len) {
    String s = new String(buf, offset, len);
    if (inName && s.equals("Mocha Java")) {
        inMochaJava = true;
        inName = false;
    } else if (inPrice) {
        System.out.println("The price of Mocha Java is: " + s);
        inMochaJava = false;
        inPrice = false;
    }
}
}

```

Once the parser has come to the Mocha Java coffee element, here is the relevant state after the following method calls:

```

next invocation of startElement -- inName is true

next invocation of characters -- inMochaJava is true

next invocation of startElement -- inPrice is true

next invocation of characters -- prints price

```

The SAX parser can perform validation while it is parsing XML data, which means that it checks that the data follows the rules specified in the XML document's schema. A SAX parser will be validating if it is created by a SAX-ParserFactory object that has had validation turned on. This is done for the SAXParserFactory object factory in the following line of code.

```
factory.setValidating(true);
```


So that the parser knows which schema to use for validation, the XML document must refer to the schema in its DOCTYPE declaration. The schema for the price list is `priceList.DTD`, so the DOCTYPE declaration should be similar to this:

```
<!DOCTYPE PriceList SYSTEM "priceList.DTD">
```

The DOM API

The Document Object Model (DOM), defined by the W3C DOM Working Group, is a set of interfaces for building an object representation, in the form of a tree, of a parsed XML document. Once you build the DOM, you can manipulate it with DOM methods such as `insert` and `remove`, just as you would manipulate any other tree data structure. Thus, unlike a SAX parser, a DOM parser allows random access to particular pieces of data in an XML document. Another difference is that with a SAX parser, you can only read an XML document, but with a DOM parser, you can build an object representation of the document and manipulate it in memory, adding a new element or deleting an existing one.

In the previous example, we used a SAX parser to look for just one piece of data in a document. Using a DOM parser would have required having the whole document object model in memory, which is generally less efficient for searches involving just a few items, especially if the document is large. In the next example, we add a new coffee to the price list using a DOM parser. We cannot use a SAX parser for modifying the price list because it only reads data.

Let's suppose that you want to add Kona coffee to the price list. You would read the XML price list file into a DOM and then insert the new coffee element, with its name and price. The following code fragment creates a `DocumentBuilderFactory` object, which is then used to create the `DocumentBuilder` object builder. The code then calls the `parse` method on `builder`, passing it the file `priceList.xml`.

```
DocumentBuilderFactory factory =  
    DocumentBuilderFactory.newInstance();  
DocumentBuilder builder = factory.newDocumentBuilder();  
Document document = builder.parse("priceList.xml");
```

At this point, `document` is a DOM representation of the price list sitting in memory. The following code fragment adds a new coffee (with the name "Kona" and a price of "13.50") to the price list document. Because we want to add the new coffee right before the coffee whose name is "Mocha Java", the first step is to get a list of the coffee elements and iterate through the list to find "Mocha Java".

Using the Node interface included in the `org.w3c.dom` package, the code then creates a Node object for the new coffee element and also new nodes for the name and price elements. The name and price elements contain character data, so the code creates a Text object for each of them and appends the text nodes to the nodes representing the name and price elements.

```
Node rootNode = document.getDocumentElement();
NodeList list = document.getElementsByTagName("coffee");

// Loop through the list.
for (int i=0; i < list.getLength(); i++) {
    thisCoffeeNode = list.item(i);
    Node thisNameNode = thisCoffeeNode.getFirstChild();
    if (thisNameNode == null) continue;
    if (thisNameNode.getFirstChild() == null) continue;
    if (! thisNameNode.getFirstChild() instanceof
        org.w3c.dom.Text) continue;

    String data = thisNameNode.getFirstChild().getNodeValue();
    if (! data.equals("Mocha Java")) continue;

    //We're at the Mocha Java node. Create and insert the new
    //element.

    Node newCoffeeNode = document.createElement("coffee");

    Node newNameNode = document.createElement("name");
    Text tnNode = document.createTextNode("Kona");
    newNameNode.appendChild(tnNode);

    Node newPriceNode = document.createElement("price");
    Text tpNode = document.createTextNode("13.50");
    newPriceNode.appendChild(tpNode);

    newCoffeeNode.appendChild(newNameNode);
    newCoffeeNode.appendChild(newPriceNode);
    rootNode.insertBefore(newCoffeeNode, thisCoffeeNode);
    break;
}
```

Note that this code fragment is a simplification in that it assumes that none of the nodes it accesses will be a comment, an attribute, or ignorable white space.

You get a DOM parser that is validating the same way you get a SAX parser that is validating: You call `setValidating(true)` on a DOM parser factory before

using it to create your DOM parser, and you make sure that the XML document being parsed refers to its schema in the DOCTYPE declaration.

XML Namespaces

All the names in a schema, which includes those in a DTD, are unique, thus avoiding ambiguity. However, if a particular XML document references multiple schemas, there is a possibility that two or more of them contain the same name. Therefore, the document needs to specify a namespace for each schema so that the parser knows which definition to use when it is parsing an instance of a particular schema.

There is a standard notation for declaring an XML Namespace, which is usually done in the root element of an XML document. In the following namespace declaration, the notation `xmlns` identifies `nsName` as a namespace, and `nsName` is set to the URL of the actual namespace:

```
<priceList xmlns:nsName="myDTD.dtd"
           xmlns:otherNsName="myOtherDTD.dtd">
...
</priceList>
```

Within the document, you can specify which namespace an element belongs to as follows:

```
<nsName:price> ...
```

To make your SAX or DOM parser able to recognize namespaces, you call the method `setNamespaceAware(true)` on your `ParserFactory` instance. After this method call, any parser that the parser factory creates will be namespace aware.

The XSLT API

XML Stylesheet Language for Transformations (XSLT), defined by the W3C XSL Working Group, describes a language for transforming XML documents into other XML documents or into other formats. To perform the transformation, you usually need to supply a style sheet, which is written in the XML Stylesheet Language (XSL). The XSL style sheet specifies how the XML data will be displayed, and XSLT uses the formatting instructions in the style sheet to perform the transformation.

JAXP supports XSLT with the `javax.xml.transform` package, which allows you to plug in an XSLT transformer to perform transformations. The subpackages have SAX-, DOM-, and stream-specific APIs that allow you to perform transformations directly from DOM trees and SAX events. The following two examples illustrate how to create an XML document from a DOM tree and how to transform the resulting XML document into HTML using an XSL style sheet.

Transforming a DOM Tree to an XML Document

To transform the DOM tree created in the previous section to an XML document, the following code fragment first creates a Transformer object that will perform the transformation.

```
TransformerFactory transFactory =  
    TransformerFactory.newInstance();  
Transformer transformer = transFactory.newTransformer();
```

Using the DOM tree root node, the following line of code constructs a DOMSource object as the source of the transformation.

```
DOMSource source = new DOMSource(document);
```

The following code fragment creates a StreamResult object to take the results of the transformation and transforms the tree into an XML file.

```
File newXML = new File("newXML.xml");  
FileOutputStream os = new FileOutputStream(newXML);  
StreamResult result = new StreamResult(os);  
transformer.transform(source, result);
```

Transforming an XML Document to an HTML Document

You can also use XSLT to convert the new XML document, `newXML.xml`, to HTML using a style sheet. When writing a style sheet, you use XML Namespaces to reference the XSL constructs. For example, each style sheet has a

root element identifying the style sheet language, as shown in the following line of code.

```
<xsl:stylesheet version="1.0" xmlns:xsl=
    "http://www.w3.org/1999/XSL/Transform">
```

When referring to a particular construct in the style sheet language, you use the namespace prefix followed by a colon and the particular construct to apply. For example, the following piece of style sheet indicates that the name data must be inserted into a row of an HTML table.

```
<xsl:template match="name">
    <tr><td>
        <xsl:apply-templates/>
    </td></tr>
</xsl:template>
```

The following style sheet specifies that the XML data is converted to HTML and that the coffee entries are inserted into a row in a table.

```
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="priceList">
        <html><head>Coffee Prices</head>
        <body>
            <table>
                <xsl:apply-templates />
            </table>
        </body>
    </html>
    </xsl:template>
    <xsl:template match="name">
        <tr><td>
            <xsl:apply-templates />
        </td></tr>
    </xsl:template>
    <xsl:template match="price">
        <tr><td>
            <xsl:apply-templates />
        </td></tr>
    </xsl:template>
</xsl:stylesheet>
```

To perform the transformation, you need to obtain an XSLT transformer and use it to apply the style sheet to the XML data. The following code fragment obtains a transformer by instantiating a `TransformerFactory` object, reading in the

style sheet and XML files, creating a file for the HTML output, and then finally obtaining the Transformer object transformer from the TransformerFactory object tFactory.

```
TransformerFactory tFactory =  
    TransformerFactory.newInstance();  
String stylesheet = "prices.xml";  
String sourceId = "newXML.xml";  
File pricesHTML = new File("pricesHTML.html");  
FileOutputStream os = new FileOutputStream(pricesHTML);  
Transformer transformer =  
    tFactory.newTransformer(new StreamSource(stylesheet));
```

The transformation is accomplished by invoking the transform method, passing it the data and the output stream.

```
transformer.transform(  
    new StreamSource(sourceId), new StreamResult(os));
```

JAX-RPC

The Building Web Services With JAX-RPC (page 269) (JAX-RPC) is the Java API for developing and using Web services.

Overview of JAX-RPC

An RPC-based Web service is a collection of procedures that can be called by a remote client over the Internet. For example, a typical RPC-based Web service is a stock quote service that takes a SOAP (Simple Object Access Protocol) request for the price of a specified stock and returns the price via SOAP.

Note: The SOAP 1.1 specification, available from <http://www.w3.org/>, defines a framework for the exchange of XML documents. It specifies, among other things, what is required and optional in a SOAP message and how data can be encoded and transmitted. JAX-RPC and JAXM are both based on SOAP.

A Web service, a server application that implements the procedures that are available for clients to call, is deployed on a server-side Web container. The container can be a standalone Web container or part of a J2EE server.

A Web service can make itself available to potential clients by describing itself in a Web Services Description Language (WSDL) document. A WSDL description is an XML document that gives all the pertinent information about a Web service, including its name, the operations that can be called on it, the parameters for those operations, and the location of where to send requests. A consumer (Web client) can use the WSDL document to discover what the service offers and how to access it. How a developer can use a WSDL document in the creation of a Web service is discussed later.

Interoperability

Perhaps the most important requirement for a Web service is that it be interoperable across clients and servers. With JAX-RPC, a client written in a language other than the Java programming language can access a Web service developed and deployed on the Java platform. Conversely, a client written in the Java programming language can communicate with a service that was developed and deployed using some other platform.

What makes this interoperability possible is JAX-RPC's support for SOAP and WSDL. SOAP defines standards for XML messaging and the mapping of data types so that applications adhering to these standards can communicate with each other. JAX-RPC adheres to SOAP standards, and is, in fact, based on SOAP messaging. That is, a JAX-RPC remote procedure call is implemented as a request-response SOAP message.

The other key to interoperability is JAX-RPC's support for WSDL. A WSDL description, being an XML document that describes a Web service in a standard way, makes the description portable. WSDL documents and their uses will be discussed more later.

Ease of Use

Given the fact that JAX-RPC is based on a remote procedure call (RPC) mechanism, it is remarkably developer friendly. RPC involves a lot of complicated infrastructure, or "plumbing," but JAX-RPC mercifully makes the underlying implementation details invisible to both the client and service developer. For example, a Web services client simply makes Java method calls, and all the internal marshalling, unmarshalling, and transmission details are taken care of automatically. On the server side, the Web service simply implements the services it offers and, like the client, does not need to bother with the underlying implementation mechanisms.

Largely because of its ease of use, JAX-RPC is the main Web services API for both client and server applications. JAX-RPC focuses on point-to-point SOAP messaging, the basic mechanism that most clients of Web services use. Although it can provide asynchronous messaging and can be extended to provide higher quality support, JAX-RPC concentrates on being easy to use for the most common tasks. Thus, JAX-RPC is a good choice for applications that wish to avoid the more complex aspects of SOAP messaging and for those that find communication using the RPC model a good fit. The more heavy-duty alternative for SOAP messaging, the Java™ API for XML Messaging (JAXM), is discussed later in this introduction.

Advanced Features

Although JAX-RPC is based on the RPC model, it offers features that go beyond basic RPC. For one thing, it is possible to send complete documents and also document fragments. In addition, JAX-RPC supports SOAP message handlers, which make it possible to send a wide variety of messages. And JAX-RPC can be extended to do one-way messaging in addition to the request-response style of messaging normally done with RPC. Another advanced feature is extensible type mapping, which gives JAX-RPC still more flexibility in what can be sent.

Using JAX-RPC

In a typical scenario, a business might want to order parts or merchandise. It is free to locate potential sources however it wants, but a convenient way is through a business registry and repository service such as a Universal Description, Discovery and Integration (UDDI) registry. Note that the Java API for XML Registries (JAXR), which is discussed later in this introduction, offers an easy way to search for Web services in a business registry and repository. Web services generally register themselves with a business registry and store relevant documents, including their WSDL descriptions, in its repository.

After searching a business registry for potential sources, the business might get several WSDL documents, one for each of the Web services that meets its search criteria. The business client can use these WSDL documents to see what the services offer and how to contact them.

Another important use for a WSDL document is as a basis for creating helper classes, used by a client to communicate with a remote service and by the server to communicate with a remote client.

A JAX-RPC runtime system converts the client's remote method call into a SOAP message and sends it to the service as an HTTP request. On the server side, the JAX-RPC runtime system receives the request, translates the SOAP message into a method call, and invokes it. After the Web service has processed the request, the runtime system goes through a similar set of steps to return the result to the client. The point to remember is that as complex as the implementation details of communication between the client and server may be, they are invisible to both Web services and their clients.

Creating a Web Service

Developing a Web service using JAX-RPC is surprisingly easy. The service itself is basically two files, an interface that declares the service's remote procedures and a class that implements those procedures. There is a little more to it, in that the service needs to be configured and deployed, but first, let's take a look at the two main components of a Web service, the interface definition and its implementation class.

The following interface definition is a simple example showing the methods a wholesale coffee distributor might want to make available to its prospective customers. Note that a service definition interface extends `java.rmi.Remote` and its methods throw a `java.rmi.RemoteException` object.

```
package coffees;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface CoffeeOrderIF extends Remote {
    public Coffee [] getPriceList()
        throws RemoteException;
    public String orderCoffee(String coffeeName, int quantity)
        throws RemoteException;
}
```

The method `getPriceList` returns an array of `Coffee` objects, each of which contains a `name` field and a `price` field. There is one `Coffee` object for each of the coffees the distributor currently has for sale. The method `orderCoffee` returns a `String` that might confirm the order or state that it is on back order.

The following example shows what the implementation might look like (with implementation details omitted). Presumably, the method `getPriceList` will query the company's database to get the current information and return the result

as an array of Coffee objects. The second method, `orderCoffee`, will also need to query the database to see if the particular coffee specified is available in the quantity ordered. If so, the implementation will set the internal order process in motion and send a reply informing the customer that the order will be filled. If the quantity ordered is not available, the implementation might place its own order to replenish its supply and notify the customer that the coffee is backordered.

```
package coffees;

public class CoffeeOrderImpl implements CoffeeOrderIF {
    public Coffee [] getPriceList() throws RemoteException; {
        . . .
    }

    public String orderCoffee(String coffeeName, int quantity)
        throws RemoteException; {
        . . .
    }
}
```

After writing the service's interface and implementation class, the developer's next step is to generate the helper classes. The final steps in creating a Web service are packaging and deployment. A Web service definition is packaged in a Web application archive (WAR). For example, the `CoffeeOrder` service could be packaged in the file `jaxrpc-coffees.war`, which makes it easy to distribute and deploy.

Coding a Client

Writing the client application for a Web service entails simply writing code that invokes the desired method. Of course, much more is required to build the remote method call and transmit it to the Web service, but that is all done behind the scenes and is invisible to the client.

The following class definition is an example of a Web services client. It creates an instance of `CoffeeOrderIF` and uses it to call the method `getPriceList`. Then it accesses the price and name fields of each `Coffee` object in the array returned by the method `getPriceList` in order to print them out.

The class `CoffeeOrderServiceImpl` is one of the classes generated by the mapping tool. It is a stub factory whose only method is `getCoffeeOrderIF`; in other words, its whole purpose is to create instances of `CoffeeOrderIF`. The instances

of `CoffeeOrderIF` that are created by `CoffeeOrderServiceImpl` are client side stubs that can be used to invoke methods defined in the interface `CoffeeOrderIF`. Thus, the variable `coffeeOrder` represents a client stub that can be used to call `getPriceList`, one of the methods defined in `CoffeeOrderIF`.

The method `getPriceList` will block until it has received a response and returned it. Because a WSDL document is being used, the JAX-RPC runtime will get the service endpoint from it. Thus, in this case, the client class does not need to specify the destination for the remote procedure call. When the service endpoint does need to be given, it can be supplied as an argument on the command line. Here is what a client class might look like:

```
package coffees;

public class CoffeeClient {
    public static void main(String[] args) {
        try {
            CoffeeOrderIF coffeeOrder = new
                CoffeeOrderServiceImpl().getCoffeeOrderIF();
            Coffee [] priceList =
                coffeeOrder.getPriceList();
            for (int i = 0; i < priceList.length; i++) {
                System.out.print(priceList[i].getName() + " ");
                System.out.println(priceList[i].getPrice());
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Invoking a Remote Method

Once a client has discovered a Web service, it can invoke one of the service's methods. The following example makes the remote method call `getPriceList`, which takes no arguments. As noted previously, the JAX-RPC runtime can determine the endpoint for the `CoffeeOrder` service (which is its URI) from its WSDL description. If a WSDL document had not been used, you would need to supply the service's URI as a command line argument. After you have compiled the file `CoffeeClient.java`, here is all you need to type at the command line to invoke its `getPriceList` method.

```
java coffees.CoffeeClient
```

The remote procedure call made by the previous line of code is a static method call. In other words, the RPC was determined at compile time. It should be noted that with JAX-RPC, it is also possible to call a remote method dynamically at run time. This can be done using either the Dynamic Invocation Interface (DII) or a dynamic proxy.

JAXM

The Web Services Messaging with JAXM (page 289) (JAXM) provides a standard way to send XML documents over the Internet from the Java platform. It is based on the SOAP 1.1 and SOAP with Attachments specifications, which define a basic framework for exchanging XML messages. JAXM can be extended to work with higher level messaging protocols, such as the one defined in the ebXML (electronic business XML) Message Service Specification, by adding the protocol's functionality on top of SOAP.

Note: The ebXML Message Service Specification is available from <http://www.oasis-open.org/committees/ebxml-msg/>. Among other things, it provides a more secure means of sending business messages over the Internet than the SOAP specifications do.

Typically, a business uses a messaging provider service, which does the behind-the-scenes work required to transport and route messages. When a messaging provider is used, all JAXM messages go through it, so when a business sends a message, the message first goes to the sender's messaging provider, then to the recipient's messaging provider, and finally to the intended recipient. It is also possible to route a message to go to intermediate recipients before it goes to the ultimate destination.

Because messages go through it, a messaging provider can take care of house-keeping details like assigning message identifiers, storing messages, and keeping track of whether a message has been delivered before. A messaging provider can also try resending a message that did not reach its destination on the first attempt at delivery. The beauty of a messaging provider is that the client using JAXM technology ("JAXM client") is totally unaware of what the provider is doing in the background. The JAXM client simply makes Java method calls, and the messaging provider in conjunction with the messaging infrastructure makes everything happen behind the scenes.

Though in the typical scenario a business uses a messaging provider, it is also possible to do JAXM messaging without using a messaging provider. In this case, the JAXM client (called a *standalone* client) is limited to sending point-to-point messages directly to a Web service that is implemented for request-response messaging. Request-response messaging is synchronous, meaning that a request is sent and its response is received in the same operation. A request-response message is sent over a `SOAPConnection` object via the method `SOAPConnection.call`, which sends the message and blocks until it receives a response. A standalone client can operate only in a client role, that is, it can only send requests and receive their responses. In contrast, a JAXM client that uses a messaging provider may act in either the client or server (service) role. In the client role, it can send requests; in the server role, it can receive requests, process them, and send responses.

Though it is not required, JAXM messaging usually takes place within a container, generally a servlet container or J2EE server. A Web service that uses a messaging provider and is deployed in a container has the capability of doing one-way messaging, meaning that it can receive a request as a one-way message and can return a response some time later as another one-way message.

Because of the features that a messaging provider can supply, JAXM can sometimes be a better choice for SOAP messaging than JAX-RPC. The following list includes features that JAXM can provide and that RPC, including JAX-RPC, does not generally provide:

- One-way (asynchronous) messaging
- Routing of a message to more than one party
- Reliable messaging with features such as guaranteed delivery

A `SOAPMessage` object represents an XML document that is a SOAP message. A `SOAPMessage` object always has a required SOAP part, and it may also have one or more attachment parts. The SOAP part must always have a `SOAPEnvelope` object, which must in turn always contain a `SOAPBody` object. The `SOAPEnvelope` object may also contain a `SOAPHeader` object, to which one or more headers can be added.

The `SOAPBody` object can hold XML fragments as the content of the message being sent. If you want to send content that is not in XML format or that is an entire XML document, your message will need to contain an attachment part in addition to the SOAP part. There is no limitation on the content in the attachment part, so it can include images or any other kind of content, including XML fragments and documents.

Getting a Connection

The first thing a JAXM client needs to do is get a connection, either a `SOAPConnection` object or a `ProviderConnection` object.

Getting a Point-to-Point Connection

A standalone client is limited to using a `SOAPConnection` object, which is a point-to-point connection that goes directly from the sender to the recipient. All JAXM connections are created by a connection factory. In the case of a `SOAPConnection` object, the factory is a `SOAPConnectionFactory` object. A client obtains the default implementation for `SOAPConnectionFactory` by calling the following line of code.

```
SOAPConnectionFactory factory =  
    SOAPConnectionFactory.newInstance();
```

The client can use `factory` to create a `SOAPConnection` object.

```
SOAPConnection con = factory.createConnection();
```

Getting a Connection to the Messaging Provider

In order to use a messaging provider, an application must obtain a `ProviderConnection` object, which is a connection to the messaging provider rather than to a specified recipient. There are two ways to get a `ProviderConnection` object, the first being similar to the way a standalone client gets a `SOAPConnection` object. This way involves obtaining an instance of the default implementation for `ProviderConnectionFactory`, which is then used to create the connection.

```
ProviderConnectionFactory pcFactory =  
    ProviderConnectionFactory.newInstance();  
ProviderConnection pcCon = pcFactory.createConnection();
```

The variable `pcCon` represents a connection to the default implementation of a JAXM messaging provider.

The second way to create a `ProviderConnection` object is to retrieve a `ProviderConnectionFactory` object that is implemented to create connections to a specific messaging provider. The following code demonstrates getting such a

ProviderConnectionFactory object and using it to create a connection. The first two lines use the Java Naming and Directory Interface™ (JNDI) API to retrieve the appropriate ProviderConnectionFactory object from the naming service where it has been registered with the name “CoffeeBreakProvider”. When this logical name is passed as an argument, the method lookup returns the ProviderConnectionFactory object to which the logical name was bound. The value returned is a Java Object, which must be narrowed to a ProviderConnectionFactory object so that it can be used to create a connection. The third line uses a JAXM method to actually get the connection.

```
Context ctx = getInitialContext();
ProviderConnectionFactory pcFactory =
    (ProviderConnectionFactory)ctx.lookup("CoffeeBreakProvider");

ProviderConnection con = pcFactory.createConnection();
```

The ProviderConnection instance con represents a connection to The Coffee Break’s messaging provider.

Creating a Message

As is true with connections, messages are created by a factory. And similar to the case with connection factories, MessageFactory objects can be obtained in two ways. The first way is to get an instance of the default implementation for the MessageFactory class. This instance can then be used to create a basic SOAPMessage object.

```
MessageFactory messageFactory = MessageFactory.newInstance();
SOAPMessage m = messageFactory.createMessage();
```

All of the SOAPMessage objects that messageFactory creates, including m in the previous line of code, will be basic SOAP messages. This means that they will have no pre-defined headers.

Part of the flexibility of the JAXM API is that it allows a specific usage of a SOAP header. For example, protocols such as ebXML can be built on top of SOAP messaging to provide the implementation of additional headers, thus enabling additional functionality. This usage of SOAP by a given standards group or industry is called a *profile*. (See the JAXM tutorial section Profiles, page 299 for more information on profiles.)

In the second way to create a `MessageFactory` object, you use the `Provider-Connection` method `createMessageFactory` and give it a profile. The `SOAPMessage` objects produced by the resulting `MessageFactory` object will support the specified profile. For example, in the following code fragment, in which `schemaURI` is the URI of the schema for the desired profile, `m2` will support the messaging profile that is supplied to `createMessageFactory`.

```
MessageFactory messageFactory2 =  
    con.createMessageFactory(<schemaURI>);  
SOAPMessage m2 = messageFactory2.createMessage();
```

Each of the new `SOAPMessage` objects `m` and `m2` automatically contains the required elements `SOAPPart`, `SOAPEnvelope`, and `SOAPBody`, plus the optional element `SOAPHeader` (which is included for convenience). The `SOAPHeader` and `SOAPBody` objects are initially empty, and the following sections will illustrate some of the typical ways to add content.

Populating a Message

Content can be added to the `SOAPPart` object, to one or more `AttachmentPart` objects, or to both parts of a message.

Populating the SOAP Part of a Message

As stated earlier, all messages have a `SOAPPart` object, which has a `SOAPEnvelope` object containing a `SOAPHeader` object and a `SOAPBody` object. One way to add content to the SOAP part of a message is to create a `SOAPHeaderElement` object or a `SOAPBodyElement` object and add an XML fragment that you build with the method `SOAPElement.addTextNode`. The first three lines of the following code fragment access the `SOAPBody` object body, which is used to create a new `SOAPBodyElement` object and add it to body. The argument passed to the `createName` method is a `Name` object identifying the `SOAPBodyElement` being added. The last line adds the XML string passed to the method `addTextNode`.

```
SOAPPart sp = m.getSOAPPart();  
SOAPEnvelope envelope = sp.getSOAPEnvelope();  
SOAPBody body = envelope.getSOAPBody();  
SOAPBodyElement bodyElement = body.addBodyElement(  
    envelope.createName("text", "hotitems",  
        "http://hotitems.com/products/gizmo");  
    bodyElement.addTextNode("some-xml-text");
```


Another way is to add content to the `SOAPPart` object by passing it a `javax.xml.transform.Source` object, which may be a `SAXSource`, `DOMSource`, or `StreamSource` object. The `Source` object contains content for the SOAP part of the message and also the information needed for it to act as source input. A `StreamSource` object will contain the content as an XML document; the `SAXSource` or `DOMSource` object will contain content and instructions for transforming it into an XML document.

The following code fragments illustrates adding content as a `DOMSource` object. The first step is to get the `SOAPPart` object from the `SOAPMessage` object. Next the code uses methods from the JAXP API to build the XML document to be added. It uses a `DocumentBuilderFactory` object to get a `DocumentBuilder` object. Then it parses the given file to produce the document that will be used to initialize a new `DOMSource` object. Finally, the code passes the `DOMSource` object `domSource` to the method `SOAPPart.setContent`.

```
SOAPPart soapPart = message.getSOAPPart();

DocumentBuilderFactory dbf=
    DocumentBuilderFactory.newInstance();
DocumentBuilder db = dbf.newDocumentBuilder();
Document doc = db.parse("file:///foo.bar/soap.xml");
DOMSource domSource = new DOMSource(doc);

soapPart.setContent(domSource);
```

Populating the Attachment Part of a Message

A `Message` object may have no attachment parts, but if it is to contain anything that is not in XML format, that content must be contained in an attachment part. There may be any number of attachment parts, and they may contain anything from plain text to image files. In the following code fragment, the content is an image in a JPEG file, whose URL is used to initialize the `javax.activation.DataHandler` object `dh`. The `Message` object `m` creates the `AttachmentPart` object `attachPart`, which is initialized with the data handler containing the URL for the image. Finally, the message adds `attachPart` to itself.

```
URL url = new URL("http://foo.bar/img.jpg");
DataHandler dh = new DataHandler(url);
AttachmentPart attachPart = m.createAttachmentPart(dh);
m.addAttachmentPart(attachPart);
```

A `SOAPMessage` object can also give content to an `AttachmentPart` object by passing an `Object` and its content type to the method `createAttachmentPart`.

```
AttachmentPart attachPart =  
    m.createAttachmentPart("content-string", "text/plain");  
m.addAttachmentPart(attachPart);
```

A third alternative is to create an empty `AttachmentPart` object and then to pass the `AttachmentPart.setContent` method an `Object` and its content type. In this code fragment, the `Object` is a `ByteArrayInputStream` initialized with a jpeg image.

```
AttachmentPart ap = m.createAttachmentPart();  
byte[] jpegData = ...;  
ap.setContent(new ByteArrayInputStream(jpegData),  
    "image/jpeg");  
m.addAttachmentPart(ap);
```

Sending a Message

Once you have populated a `SOAPMessage` object, you are ready to send it. A standalone client uses the `SOAPConnection` method `call` to send a message. This method sends the message and then blocks until it gets back a response. The arguments to the method `call` are the message being sent and a `URL` object that contains the URL specifying the endpoint of the receiver..

```
SOAPMessage response =  
    soapConnection.call(message, endpoint);
```

An application that is using a messaging provider uses the `ProviderConnection` method `send` to send a message. This method sends the message asynchronously, meaning that it sends the message and returns immediately. The response, if any, will be sent as a separate operation at a later time. Note that this method takes only one parameter, the message being sent. The messaging provider will use header information to determine the destination.

```
providerConnection.send(message);
```

JAXR

The Publishing and Discovering Web Services with JAXR, page 347 (JAXR) provides a convenient way to access standard business registries over the Internet. Business registries are often described as electronic yellow pages because they contain listings of businesses and the products or services the businesses offer. JAXR gives developers writing applications in the Java programming language a uniform way to use business registries that are based on open standards (such as ebXML) or industry consortium-led specifications (such as UDDI).

Businesses can register themselves with a registry or discover other businesses with which they might want to do business. In addition, they can submit material to be shared and search for material that others have submitted. Standards groups have developed schemas for particular kinds of XML documents, and two businesses might, for example, agree to use the schema for their industry's standard purchase order form. Because the schema is stored in a standard business registry, both parties can use JAXR to access it.

Registries are becoming an increasingly important component of Web services because they allow businesses to collaborate with each other dynamically in a loosely coupled way. Accordingly, the need for JAXR, which enables enterprises to access standard business registries from the Java programming language, is also growing.

Using JAXR

The following sections give examples of two of the typical ways a business registry is used. They are meant to give you an idea of how to use JAXR rather than to be complete or exhaustive.

Registering a Business

An organization that uses the Java platform for its electronic business would use JAXR to register itself in a standard registry. It would supply its name, a description of itself, and some classification concepts to facilitate searching for it. This is shown in the following code fragment, which first creates the `RegistryService` object `rs` and then uses it to create the `BusinessLifecycleManager` object `lcm` and the `BusinessQueryManager` object `bqm`. The business, a chain of coffee houses called The Coffee Break, is represented by the `Organization` object `org`, to which The Coffee Break adds its name, a description of itself, and its classifi-

cation within the North American Industry Classification System (NAICS). Then `org`, which now contains the properties and classifications for The Coffee Break, is added to the `Collection` object `orgs`. Finally, `orgs` is saved by `lcm`, which will manage the life cycle of the `Organization` objects contained in `orgs`.

```
RegistryService rs = connection.getRegistryService();

BusinessLifeCycleManager lcm =
    rs.getBusinessLifeCycleManager();
BusinessQueryManager bqm =
    rs.getBusinessQueryManager();

Organization org = lcm.createOrganization("The Coffee Break");
org.setDescription(
    "Purveyor of only the finest coffees. Established 1895");

ClassificationScheme cScheme =
    bqm.findClassificationSchemeByName("ntis-gov:naics");

Classification classification =
    (Classification)lcm.createClassification(cScheme,
        "Snack and Nonalcoholic Beverage Bars", "722213");

Collection classifications = new ArrayList();
classifications.add(classification);

org.addClassifications(classifications);
Collection orgs = new ArrayList();
orgs.add(org);
lcm.saveOrganizations(orgs);
```

Searching a Registry

A business can also use JAXR to search a registry for other businesses. The following code fragment uses the `BusinessQueryManager` object `bqm` to search for The Coffee Break. Before `bqm` can invoke the method `findOrganizations`, the code needs to define the search criteria to be used. In this case, three of the possible six search parameters are supplied to `findOrganizations`; because `null` is supplied for the third, fifth, and sixth parameters, those criteria are not used to limit the search. The first, second, and fourth arguments are all `Collection` objects, with `findQualifiers` and `namePatterns` being defined here. The only element in `findQualifiers` is a `String` specifying that no organization be returned unless its name is a case-sensitive match to one of the names in the `namePatterns` parameter. This parameter, which is also a `Collection` object

with only one element, says that businesses with “Coffee” in their names are a match. The other `Collection` object is `classifications`, which was defined when The Coffee Break registered itself. The previous code fragment, in which the industry for The Coffee Break was provided, is an example of defining classifications.

```
BusinessQueryManager bqm = rs.getBusinessQueryManager();

//Define find qualifiers
Collection findQualifiers = new ArrayList();
findQualifiers.add(FindQualifier.CASE_SENSITIVE_MATCH);
Collection namePatterns = new ArrayList();
namePatterns.add("%Coffee%"); // Find orgs with name containing
//'Coffee'

//Find using only the name and the classifications
BulkResponse response = bqm.findOrganizations(findQualifiers,
    namePatterns, null, classifications, null);
Collection orgs = response.getCollection();
```

JAXR also supports using an SQL query to search a registry. This is done using a `DeclarativeQueryManager` object, as the following code fragment demonstrates.

```
DeclarativeQueryManager dqm = rs.getDeclarativeQueryManager();
Query query = dqm.createQuery(Query.QUERY_TYPE_SQL,
    "SELECT id FROM RegistryEntry WHERE name LIKE %Coffee% " +
    "AND majorVersion >= 1 AND " +
    "(majorVersion >= 2 OR minorVersion >= 3)");
BulkResponse response2 = dqm.executeQuery(query);
```

The `BulkResponse` object `response2` will contain a value for `id` (a uuid) for each entry in `RegistryEntry` that has “Coffee” in its name and that also has a version number of 1.3 or greater.

To ensure interoperable communication between a JAXR client and a registry implementation, the messaging is done using JAXM. This is done completely behind the scenes, so as a user of JAXR, you are not even aware of it.

Sample Scenario

The following scenario is an example of how the Java APIs for XML might be used and how they work together. Part of the richness of the Java APIs for XML

is that in many cases they offer alternate ways of doing something and thus let you tailor your code to meet individual needs. This section will point out some instances in which an alternate API could have been used and will also give the reasons why one API or the other might be a better choice.

Scenario

Suppose that the owner of a chain of coffee houses, called The Coffee Break, wants to expand by selling coffee online. He instructs his business manager to find some new coffee suppliers, get their wholesale prices, and then arrange for orders to be placed as the need arises. The Coffee Break can analyze the prices and decide which new coffees it wants to carry and which companies it wants to buy them from.

Discovering New Distributors

The business manager assigns the task of finding potential new sources of coffee to the company's software engineer. She decides that the best way to locate new coffee suppliers is to search a Universal Description, Discovery, and Integration (UDDI) registry, where The Coffee Break has already registered itself.

The engineer uses JAXR to send a query searching for wholesale coffee suppliers. The JAXR implementation uses JAXM behind the scenes to send the query to the registry, but this is totally transparent to the engineer.

The UDDI registry will receive the query and apply the search criteria transmitted in the JAXR code to the information it has about the organizations registered with it. When the search is completed, the registry will send back information on how to contact the wholesale coffee distributors that met the specified criteria. Although the registry uses JAXM behind the scenes to transmit the information, the response the engineer gets back is JAXR code.

Requesting Price Lists

The engineer's next step is to request price lists from each of the coffee distributors. She has obtained a WSDL description for each one, which tells her the procedure to call to get prices and also the URI where the request is to be sent. Her code makes the appropriate remote procedure calls using JAX-RPC API and gets back the responses from the distributors. The Coffee Break has been doing business with one distributor for a long time and has made arrangements with it to

exchange JAXM messages using agreed-upon XML schemas. Therefore, for this distributor, the engineer's code uses JAXM API to request current prices, and the distributor returns the price list in a JAXM message.

Comparing Prices and Ordering Coffees

Upon receiving the response to her request for prices, the engineer processes the price lists using SAX. She uses SAX rather than DOM because for simply comparing prices, it is more efficient. (To modify the price list, she would have needed to use DOM.) After her application gets the prices quoted by the different vendors, it compares them and displays the results.

When the owner and business manager decide which suppliers to do business with, based on the engineer's price comparisons, they are ready to send orders to the suppliers. The orders to new distributors are sent via JAX-RPC; orders to the established distributor are sent via JAXM. Each supplier, whether using JAX-RPC or JAXM, will respond by sending a confirmation with the order number and shipping date.

Selling Coffees on the Internet

Meanwhile, The Coffee Break has been preparing for its expanded coffee line. It will need to publish a price list/order form in HTML for its Web site. But before that can be done, the company needs to determine what prices it will charge. The engineer writes an application that will multiply each wholesale price by 135% to arrive at the price that The Coffee Break will charge. With a few modifications, the list of retail prices will become the online order form.

The engineer uses JavaServer Pages (JSP) technology to create an HTML order form that customers can use to order coffee online. From the JSP page, she gets the name and price of each coffee, and then she inserts them into an HTML table on the JSP page. The customer enters the quantity of each coffee desired and clicks the "Submit" button to send the order.

Conclusion

Although this scenario is simplified for the sake of brevity, it illustrates how XML technologies can be used in the world of Web services. With the availability of the Java APIs for XML and the J2EE platform, creating Web services and writing applications that use them have both gotten easier.

Chapter 12 demonstrates a simple implementation of this scenario.

Building Web Services With JAX-RPC

Dale Green

JAX-RPC stands for Java API for XML-based RPC. It's an API for building Web services and clients that use remote procedure calls (RPC) and XML. Often used in a distributed client/server model, an RPC mechanism enables clients to execute procedures on other systems.

In JAX-RPC, a remote procedure call is represented by an XML-based protocol such as SOAP. The SOAP specification defines envelope structure, encoding rules, and a convention for representing remote procedure calls and responses. These calls and responses are transmitted as SOAP messages over HTTP. In this release, JAX-RPC relies on SOAP 1.1 and HTTP 1.1.

Although JAX-RPC relies on complex protocols, the API hides this complexity from the application developer. On the server side, the developer specifies the remote procedures by defining methods in an interface written in the Java programming language. The developer also codes one or more classes that implement those methods. Client programs are also easy to code. A client creates a *proxy*, a local object representing the service, and then simply invokes methods on the proxy.

With JAX-RPC, clients and Web services have a big advantage—the platform independence of the Java programming language. In addition, JAX-RPC is not restrictive: a JAX-RPC client can access a Web service that is not running on the Java platform and vice versa. This flexibility is possible because JAX-RPC uses technologies defined by the World Wide Web Consortium (W3C): HTTP, SOAP,

and the Web Service Description Language (WSDL). WSDL specifies an XML format for describing a service as a set of endpoints operating on messages.

If you're new to the Java API for XML-based RPC (JAX-RPC), this chapter is the place to start. The chapter starts by listing the supported types and then shows you how to build a simple Web service and three types of clients.

Types Supported By JAX-RPC

Behind the scenes, JAX-RPC maps types of the Java programming language to XML/WSDL definitions. For example, JAX-RPC maps the `java.lang.String` class to the `xsd:string` XML data type. Application developers don't need to know the details of these mappings, but they should be aware that not every class in the Java 2 Standard Edition (J2SE™) can be used as a method parameter or return type in JAX-RPC.

J2SE SDK Classes

JAX-RPC supports the following J2SE SDK classes:

```
java.lang.Boolean
java.lang.Byte
java.lang.Double
java.lang.Float
java.lang.Integer
java.lang.Long
java.lang.Short
java.lang.String

java.math.BigDecimal
java.math.BigInteger

java.util.Calendar
java.util.Date
```

This release of JAX-RPC also supports several implementation classes of the `java.util.Collection` interface. See Table 9–1.

Table 9–1 Supported Classes of the Java Collections Framework

<code>java.util.Collection</code> Interface	Implementation Classes
List	ArrayList LinkedList Stack Vector
Map	HashMap Hashtable Properties TreeMap
Set	HashSet TreeSet

Primitives

JAX-RPC supports the following primitive types of the Java programming language:

- `boolean`
- `byte`
- `double`
- `float`
- `int`
- `long`
- `short`

Arrays

JAX-RPC also supports arrays with members of supported JAX-RPC types. Examples of supported arrays are `int[]` and `String[]`. Multidimensional arrays, such as `BigDecimal[][]`, are also supported.

Application Classes

JAX-RPC also supports classes that you've written for your applications. In an order processing application, for example, you might provide classes named `Order`, `LineItem`, and `Product`. The JAX-RPC Specification refers to such classes as *value types*, because their values (or states) may be passed between clients and remote services as method parameters or return values.

To be supported by JAX-RPC, an application class must conform to the following rules:

- It must have a public default constructor.
- It must not implement (either directly or indirectly) the `java.rmi.Remote` interface.
- Its fields must be supported JAX-RPC types.

The class may contain public, private, or protected fields. For its value to be passed (or returned) during a remote call, a field must meet these requirements:

- A public field cannot be `final` or `transient`.
- A non-public field must have corresponding getter and setter methods.

JavaBeans Components

JAX-RPC also supports JavaBeans components, which must conform to the same set of rules as application classes. In addition, a JavaBeans component must have a getter and setter method for each bean property. The type of the bean property must be a supported JAX-RPC type. For an example, see the section *JavaBeans Components* (page 411).

Creating a Web Service with JAX-RPC

This section describes how to build and deploy a simple Web service named `MyHelloService` using Sun ONE Studio 4 and Sun ONE Application Server 7. For the sake of brevity, instructions in this section refer to Sun ONE Studio 4 as the "IDE." After you've deployed `MyHelloService`, you can access it by the clients discussed in *Creating Web Service Clients with JAX-RPC* (page 276).

Note: The instructions that follow were written for Sun ONE Studio 4 update 1, Enterprise Edition. If you are running a later version, then you should refer to the

Sun ONE Studio Programming Series for up to date instructions. (See Further Information, page 286.)

These are the basic steps for creating a Web service:

1. Code and compile a class that implements the service's methods.
2. Create the service: New→Web Services→Web Service
3. Generate the service's helper classes and WSDL file: Right-click the service node and choose Generate Web Service.
4. Deploy the service: Right-click the service node and choose Deploy.

The sections that follow cover these steps in greater detail.

Verifying the IDE Settings

Before building `MyHelloService`, check the following.

- The Sun ONE Application Server 7 has been installed.
 - a. In the Runtime pane of the Explorer, choose Server Registry→Installed Servers.
 - b. Verify that the Sun One Application Server 7 is listed below Installed Servers. If it is not listed, then follow the instructions in Deploying Web Modules, page 33.
- The Sun ONE Application Server 7 is the default server.
 - a. In the Runtime pane of the Explorer, expand Server Registry→Default Servers.
 - b. Verify that beneath Default Servers there is a node for Web Tier Applications: `server1(host:port)`.
 - c. If this node is not displayed, then follow the instructions in the Sun ONE Studio 4, Enterprise Edition for Java Tutorial.
- You know the server instance port number.
 - a. Start the Admin Console of the Sun ONE Application Server 7.
 - b. in the console's left pane, select Domains.
 - c. In the tree of the left pane, expand the App Server Instances node and choose `server1`.

- d. Note the HTTP Port number in the right pane. In most installations, the default port number is 80. In a later section (Specifying the SOAP RPC URL, page 275), you will specify this number for MyHelloService.

Creating MyHelloService

1. In the IDE, mount the file system at `<INSTALL>/examples/jaxrpc`.
2. In the Explorer, expand the file system you just mounted.
3. Expand the `helloservice` package.

This package will contain all of the files for MyHelloService. If this is the first time you've gone through these instructions, then `helloservice` will contain a single file: the source code for `SimpleGreeter`.

4. Right-click `SimpleGreeter` and choose Open.

The Source Editor displays `SimpleGreeter.java`, which has two methods: `sayHello` and `sayGoodbye`. In a later step you will assign these methods to MyHelloService. At runtime, the service's remote clients will be able to invoke these methods.

5. Right-click the Source Editor and choose Compile.
6. Close the Output window and the Source Editor.
7. Right-click the `helloservice` package and choose `New→Web Services→Web Service`.

The Web Service pane of the New wizard appears.

8. In the wizard's Specify Web Service pane, do the following.
 - a. In the Name field, enter `MyHelloService`.
 - b. In the Package field, enter `helloservice`.
 - c. For the Create From buttons, choose Java Methods.
 - d. For the Architecture buttons, choose Web centric.
 - e. Click Next.
9. In the wizard's Select Methods pane, do the following.
 - a. Expand the nodes in the `helloservice` package until you see the methods beneath the `SimpleGreeter` class.
 - b. Choose the `sayHello` and `sayGoodbye` methods.
 - c. Click Finish.

In the Explorer, a Web service node (a blue sphere in a cube) for `MyHelloService` appears.

Specifying the SOAP RPC URL

1. Right-click the `MyHelloService` node, choose Properties, and examine the SOAP RPC URL property.

This URL is sometimes called the service endpoint address. Remote clients of the service use this URL to locate the service. In a later step, you'll see that the URL is written to the service's WSDL file.

The SOAP RPC URL property has the following syntax:

```
http://<host>:<port>/<web-context>/<url-pattern>
```

The `<host>` is the name of the computer that is running the Web server and `<port>` is the server's port number. The `<web-context>` (sometimes called the context root), is a name that is mapped to the document root of the servlet that implements the Web service. The `<url-pattern>` is an arbitrary string which may contain forward slashes, for example, `my/app/account`. The `<url-pattern>` allows you to further qualify the URL. By default, the IDE assigns the name of the Web service (`MyHelloService`) to the `<web-context>` and `<url-pattern>` elements.

2. For the SOAP RPC URL property, make sure that the port number matches the value you noted in Verifying the IDE Settings (page 273). If the host name is `localhost` and the port number is 80, then the SOAP RPC URL property should be:

```
http://localhost:80/MyHelloService/MyHelloService
```

3. If necessary, change the port number of the SOAP RPC URL.
4. Close the Properties sheet.

Generating the Service's Helper Classes and WSDL File

1. Right-click the `MyHelloService` Web service and choose Generate Web Service.

The IDE creates the service's helper classes, placing them in the `MyHelloServiceGen` package and also creates the `MyHelloService` WSDL file.

In the IDE's Explorer, a WSDL file icon appears as a blue sphere in the lower left corner of a rectangle.

A WSDL file is an XML document that describes a particular service. WSDL files are important because they decouple the service and client development processes. A service provider makes its WSDL file available to client developers. Using an IDE, a client developer specifies the WSDL file and generates the runtime classes needed by the client program. (See *Building and Running the StaticStubHello Client*, page 278.) Note that the client developer has access to the service's WSDL, but not to the service's libraries or source code.

2. Right-click the `MyHelloService` WSDL and choose **Open**.
3. In the Source Editor, scroll down to the bottom of the WSDL file.
The `location` attribute of the `soap:address` element should match the property in Specifying the SOAP RPC URL, step 2.
4. Close the Source Editor.

Deploying MyHelloService

1. Right-click the `MyHelloService` Web service node and choose **Deploy**.
2. To verify the deployment, do the following.
 - a. Start the Admin Console of the Sun One Application Server 7.
 - b. In the left pane of the Admin Console, expand these nodes: **App Server Instances**→**server1**→**Web Apps**.

The `MyHelloService` node should appear beneath **Web Apps**.

Creating Web Service Clients with JAX-RPC

This section shows how to create and run these types of clients:

- Static stub
- Dynamic proxy
- Dynamic invocation interface (DII)

When you run these client examples, they will access the `MyHelloService` that you deployed in the preceding section.

Static Stub Client Example

`StaticStubHello` is a stand-alone program that calls the `sayHello` and `sayGoodbye` methods of `MyHelloService`. It makes this call through a *stub*, a local object which acts as a proxy for the remote service. Because this stub is created before runtime (by the IDE), it is called a *static stub*.

StaticStubHello Source Code

Before it can invoke the remote methods on the stub, `StaticStubHello` performs these steps:

1. Creates a `Stub` object named `stub`:

```
return (Stub)
    new MyHelloService_Impl().getMyHelloServiceRPCPort();
```

The program gets the `Stub` object by invoking a private method named `createProxy`. Note that the code in this method is implementation-specific and may not be portable because it relies on the `MyHelloService_Impl` object. The `MyHelloService_Impl` class is created by the IDE in when you choose the `Generate Client Proxy` menu item in `Building and Running the StaticStubHello Client` (page 278).

2. Casts `stub` to the service definition interface, `MyHelloServiceRPC`:

```
MyHelloServiceRPC hello = (MyHelloServiceRPC)stub;
```

A service definition interface declares the methods that a remote client may invoke on the service. In this example, the interface (`MyHelloServiceRPC`) defines the `sayHello` and `sayGoodbye` methods. The IDE creates the `MyHelloServiceRPC` class file when you choose the `Generate Client Proxy` menu item. The IDE gets the name `MyHelloServiceRPC` from the WSDL file, which was created in `Generating the Service's Helper Classes and WSDL File` (page 275). When the IDE created the WSDL file, it constructed the name of the service definition interface by appending `RPC` to the service name (`MyHelloService`).

Here is the full source code listing for the `StaticStubHello` client:

```
package staticstub;

import javax.xml.rpc.Stub;
import staticstub.MyStaticGenClient.MyHelloService_Impl;
import staticstub.MyStaticGenClient.MyHelloServiceRPC;

public class StaticStubHello {
    public static void main(String[] args) {
        try {
            Stub stub = createProxy();
            MyHelloServiceRPC hello = (MyHelloServiceRPC)stub;
            System.out.println(hello.sayHello("Duke"));
            System.out.println(hello.sayGoodbye("Jake"));
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    private static Stub createProxy() {
        // Note: MyHelloService_Impl is implementation-specific.
        return (Stub)
        (new MyHelloService_Impl().getMyHelloServiceRPCPort());
    }
}
```

Building and Running the StaticStubHello Client

These are the basic steps for building and running the client:

1. Create the client: New→Web Services→Web Service Client.
2. Generate the client's runtime classes: Right-click the client node and choose Generate Client Proxy.
3. Right-click the client program and choose Execute.

The detailed steps follow:

1. In the Explorer, make sure that the `MyHelloService` WSDL resides in the `helloservice` package.

In a previous section, Creating `MyHelloService` (page 274), the IDE generated the WSDL file. Later in this section, the IDE reads the WSDL file for information it needs to create runtime classes for the client.

2. Right-click the `staticstub` package and choose File→New→Web Services→Web Service Client.

The Web Service Client pane of the New wizard appears.

3. In the wizard's Specify Web Service Client pane, do the following:
 - a. In the Name field enter `MyStatic`.
 - b. In the Package field, enter `staticstub`.
 - c. For the Create From buttons, choose Local WSDL File.
 - d. Click Next.
4. In the wizard's Select Local WSDL File pane, choose the `MyHelloService WSDL` of the `helloservice` package.
5. Click Finish.

The `MyStatic` client node appears in the Explorer.

6. In Explorer, right-click the `MyStatic` client node and choose Generate Client Proxy.

This action creates the `MyStatic$Documents` and `MyStaticGenClient` packages. This example will not use the `MyStatic$Document` package, which contains JSP pages for testing the service.

The `MyStaticGenClient` package contains the stub class, serializer classes, and other helper classes required by the client at runtime. This package also contains the `MyHelloServiceRPC` and `MyHelloService_Impl` classes. Because these classes are referenced in the client's source code, they must be generated before the client is compiled. (See the section `StaticStubHello` Source Code, page 277).

7. Right-click `StaticStubHello` and choose Execute.

The IDE compiles and runs the program. The Output window should display these lines:

```
Hello Duke
Goodby Jake
```

In this example, you've run the `StaticStubHello` client from within the IDE, which can locate the runtime classes with its default classpath. If you were to run the client outside of the IDE, you'd want to create a JAR file containing the runtime classes of the `MyStaticGenClient` package.

Dynamic Proxy Client Example

The client in the preceding section used a static stub for the proxy. In contrast, the client example in this section, `DynamicProxyHello`, calls a remote procedure through a *dynamic proxy*, an object created at runtime that represents the Web service. Although the source code for the `StaticStubHelloClient` example relied on an implementation-specific class, but the `DynamicProxyHello` code does not have this limitation. (However, the `DynamicProxyHello` client does rely on implementation-specific runtime classes that are generated by the IDE.)

DynamicProxyHello Source Code

The `DynamicProxyHello` program constructs the dynamic proxy as follows:

1. Creates a `Service` object named `helloService`:

```
Service helloService =  
    serviceFactory.createService(helloWsdUrl,  
    new QName(namespaceUri, serviceName));
```

A `Service` object is a factory for proxies. To create the `Service` object (`helloService`), the program calls the `createService` method on another type of factory, a `ServiceFactory` object.

The `createService` method has two parameters, the URL of the WSDL file and a `QName` object. In this example, the URL of the WSDL file points to the WSDL that has been deployed with `MyHelloService`:

```
http://localhost:80/MyHelloService/MyHelloService?WSDL
```

A `QName` object is a tuple that represents an XML qualified name. The tuple is composed of a namespace URI and the local part of the qualified name. In the `QName` parameter of the `createService` invocation, the local part is the service name, `MyHelloService`.

2. From `helloService`, creates a proxy (`myProxy`) with a type of the service definition interface (`MyHelloServiceRPC`):

```
MyHelloServiceRPC myProxy =  
    (MyHelloServiceRPC) helloService.getPort(  
    new QName(namespaceUri, portName),  
    MyHelloServiceRPC.class);
```

The `helloService` object is a factory for dynamic proxies. To create `myProxy`, the program calls the `getPort` method of `helloService`. This method has two parameters: a `QName` object that specifies the port name

and a `java.lang.Class` object for the service definition interface. The port name, `MyHelloServiceRPCPort`, is specified by the WSDL file.

When the IDE creates the WSDL, it constructs the port name by appending `RPCPort` to the service name (`MyHelloService`) that you enter in the Specify Web Service pane of the New wizard (See *Creating MyHelloService*, page 274.) The service definition interface, `MyHelloServiceRPC`, is created by the IDE when you choose the *Generate Client Proxy* menu item.

The source code for the `DynamicProxyHello` client follows:

```
package dynamicproxy;

import java.net.URL;
import javax.xml.rpc.Service;
import javax.xml.rpc.JAXRPCException;
import javax.xml.namespace.QName;
import javax.xml.rpc.ServiceFactory;
import dynamicproxy.MyDynamicGenClient.MyHelloServiceRPC;

public class DynamicProxyHello {

    public static void main(String[] args) {
        try {

            String urlString =
                "http://localhost:80/MyHelloService/MyHelloService?WSDL";
            String namespaceUri = "urn:MyHelloService/wsd1";
            String serviceName = "MyHelloService";
            String portName = "MyHelloServiceRPCPort";

            URL helloWsdUrl = new URL(urlString);

            ServiceFactory serviceFactory =
                ServiceFactory.newInstance();

            Service helloService =
                serviceFactory.createService(helloWsdUrl,
                    new QName(namespaceUri, serviceName));

            MyHelloServiceRPC myProxy =
                (MyHelloServiceRPC) helloService.getPort(
                    new QName(namespaceUri, portName),
                    MyHelloServiceRPC.class);

            System.out.println(myProxy.sayHello("Buzz"));
```

```
        } catch (Exception ex) {  
            ex.printStackTrace();  
        }  
    }  
}
```

Building and Running the DynamicProxyHello Client

Before performing the steps in this section, you must first create and deploy the `MyHelloService` as described in [Creating a Web Service with JAX-RPC](#) (page 272). The steps for building and running the `DynamicProxyHello` client are the same as those described in [Building and Running the StaticStubHello Client](#) (page 278), with the following exceptions:

1. In the New wizard's Specify Web Service Client pane, enter `MyDynamic` in the Name field and `dynamicproxy` in the Package field.
2. When you execute the `DynamicProxyHello` client, the Output window should display this line:

```
Hello Buzz
```

Dynamic Invocation Interface (DII) Client Example

With the dynamic invocation interface (DII), a client can call a remote procedure even if the signature of the remote procedure or the name of the service are unknown until runtime. In contrast to a static stub or dynamic proxy client, a DII client does not require runtime classes generated by the IDE. However, as you'll see in the following section, the source code for a DII client is more complicated than the code of the other two types of clients.

Note: This example is for advanced users who are familiar with WSDL documents. (See [Further Information](#), page 286.)

DIIHello Source Code

The DIIHello program performs these steps:

1. Creates a Service object.

```
Service service =
    factory.createService(new QName(qnameService));
```

To get a Service object, the program invokes the createService method of a ServiceFactory object. The parameter of the createService method is a QName object that represents the name of the service, MyHelloService. The WSDL file specifies this name as follows:

```
<service name="MyHelloService">
```

2. From the Service object, creates a Call object:

```
QName port = new QName(qnamePort);
Call call = service.createCall(port);
```

A Call object supports the dynamic invocation of the remote procedures of a service. To get a Call object, the program invokes the Service object's createCall method. The parameter of createCall is a QName object that represents the service definition interface, MyHelloServiceRPC. In the WSDL file, the name of this interface is designated by the portType element:

```
<portType name="MyHelloServiceRPC">
```

3. Sets the target endpoint address of the Call object:

```
call.setTargetEndpointAddress(endpoint);
```

This address is the URL of the service. (For a static stub client, the IDE refers to the endpoint address as the SOAP RPC URL.) In the WSDL file, this address is specified by the <soap:address> element:

```
<service name="MyHelloService">
  <port name="MyHelloServiceRPCPort"
    binding="tns:MyHelloServiceRPCBinding">
    <soap:address
      location="http://localhost:80/MyHelloService/MyHelloService"/>
    </port>
  </service>
```

4. Sets these properties on the Call object:

```
SOAPACTION_USE_PROPERTY
SOAPACTION_URI_PROPERTY
ENCODING_STYLE_PROPERTY
```

To learn more about these properties, refer to the SOAP and WSDL documents listed in Further Information (page 286).

5. Specifies the method's return type, name, and parameter:

```
QName QNAME_TYPE_STRING = new QName(NS_XSD, "string");
call.setReturnType(QNAME_TYPE_STRING);

call.setOperationName(new QName(BODY_NAMESPACE_VALUE,
    "sayHello"));

call.addParameter("String_1", QNAME_TYPE_STRING,
    ParameterMode.IN);
```

To specify the return type, the program invokes the `setReturnType` method on the `Call` object. The parameter of `setReturnType` is a `QName` object that represents an XML string type.

The program designates the method name by invoking the `setOperationName` method with a `QName` object that represents `sayHello`.

To indicate the method parameter, the program invokes the `addParameter` method on the `Call` object. The `addParameter` method has three arguments: a `String` for the parameter name (`String_1`), a `QName` object for the XML type, and a `ParameterMode` object to indicate the passing mode of the parameter (`IN`).

6. Invokes the remote method on the `Call` object:

```
String[] params = { "Murphy" };
String result = (String)call.invoke(params);
```

The program assigns the parameter value (`Murphy`) to a `String` array (`params`) and then executes the `invoke` method with the `String` array as an argument.

Here is the source code for the `DIIHello` client:

```
package dii;

import javax.xml.rpc.Call;
import javax.xml.rpc.Service;
import javax.xml.rpc.JAXRPCException;
import javax.xml.namespace.QName;
import javax.xml.rpc.ServiceFactory;
import javax.xml.rpc.ParameterMode;
```



```
public class DIIHello {

    private static String qnameService = "MyHelloService";
    private static String qnamePort = "MyHelloServiceRPC";
    private static String endpoint =
        "http://localhost:80/MyHelloService/MyHelloService";

    private static String BODY_NAMESPACE_VALUE =
        "urn:MyHelloService/wsd1";
    private static String ENCODING_STYLE_PROPERTY =
        "javax.xml.rpc.encodingstyle.namespace.uri";
    private static String NS_XSD =
        "http://www.w3.org/2001/XMLSchema";
    private static String URI_ENCODING =
        "http://schemas.xmlsoap.org/soap/encoding/";

    public static void main(String[] args) {
        try {

            ServiceFactory factory =
                ServiceFactory.newInstance();
            Service service =
                factory.createService(new QName(qnameSer-
vice));

            QName port = new QName(qnamePort);

            Call call = service.createCall(port);
            call.setTargetEndpointAddress(endpoint);

            call.setProperty(Call.SOAPACTION_USE_PROPERTY,
                new Boolean(true));
            call.setProperty(Call.SOAPACTION_URI_PROPERTY,
                "");
            call.setProperty(ENCODING_STYLE_PROPERTY,
                URI_ENCODING);
            QName QNAME_TYPE_STRING = new QName(NS_XSD,
                "string");
            call.setReturnType(QNAME_TYPE_STRING);
```

```

call.setOperationName(new
QName(BODY_NAMESPACE_VALUE,
      "sayHello"));
call.addParameter("String_1", QName_TYPE_STRING,
      ParameterMode.IN);
String[] params = { "Murphy" };

String result = (String)call.invoke(params);
System.out.println(result);

    } catch (Exception ex) {
        ex.printStackTrace();
    }
}
}

```

Building and Running the DIIHello Client

Because a DII client does not require generated runtime classes, the procedures for building and running DIIHello are simple.

1. Make sure you've followed the instructions in *Deploying MyHelloService* (page 276).
2. In the Explorer, expand the `dii` package.
3. Right-click `DIIHello` and choose **Execute**.

The Output window should display this line:

```
Hello Murphy
```

Further Information

For more information about JAX-RPC and related technologies, refer to the following:

- Java API for XML-based RPC 1.0 Specification
<http://java.sun.com/xml/downloads/jaxrpc.html>
- JAX-RPC Home
<http://java.sun.com/xml/jaxrpc/index.html>
- Simple Object Access Protocol (SOAP) 1.1 W3C Note

<http://www.w3.org/TR/SOAP/>

- Web Services Description Language (WSDL) 1.1 W3C Note
<http://www.w3.org/TR/wsdl>

To learn more about Sun ONE Studio 4, see the following:

- *Sun™ ONE Studio 4, Enterprise Edition for Java™ Tutorial*
<http://forte.sun.com/ffj/documentation/s1s41/s1seetut.pdf>
- *Building Web Services (Sun ONE Studio 4 Programming Series)*
<http://forte.sun.com/ffj/documentation/s1s41/websrvcs.pdf>
- StockApp Example
<http://www.sun.com/software/sundev/jde/examples/index.html>

Web Services Messaging with JAXM

Maydene Fisher and Kim Haase

THE Java API for XML Messaging (JAXM) makes it possible for developers to do XML messaging using the Java platform. By simply making method calls using the JAXM API, you can create and send XML messages over the Internet. This chapter will help you learn how to use the JAXM API.

In addition to stepping you through how to use the JAXM API, this chapter gives instructions for running the sample JAXM applications included with this tutorial as a way to help you get started. You may prefer to go through both the overview and tutorial before running the samples to make it easier to understand what the sample applications are doing, or you may prefer to explore the samples first. The overview gives some of the conceptual background behind the JAXM API to help you understand why certain things are done the way they are. The tutorial shows you how to use the basic JAXM API, giving examples and explanations of the more commonly used features. Finally, the code examples in the last part of the tutorial show how to build an application.

The Structure of the JAXM API

The JAXM API conforms to the Simple Object Access Protocol (SOAP) 1.1 specification and the SOAP with Attachments specification. The complete JAXM API is presented in two packages:

- **javax.xml.soap** — the package defined in the SOAP with Attachments API for Java (SAAJ) 1.1 specification. This is the basic package for SOAP messaging, which contains the API for creating and populating a SOAP message. This package has all the API necessary for sending request-response messages. (Request-response messages are explained in `SOAPConnection`, page 296.)

The current version is SAAJ 1.1_02.

- **javax.xml.messaging** — the package defined in the JAXM 1.1 specification. This package contains the API needed for using a messaging provider and thus for being able to send one-way messages. (One-way messages are explained in `ProviderConnection`, page 297.)

The current version is JAXM 1.1_01.

Originally, both packages were defined in the JAXM 1.0 specification. The `javax.xml.soap` package was separated out and expanded into the SAAJ 1.1 specification so that now it has no dependencies on the `javax.xml.messaging` package and thus can be used independently. The SAAJ API also makes it easier to create XML fragments, which are especially helpful for developing JAX-RPC implementations.

The `javax.xml.messaging` package, defined in the JAXM 1.1 specification, maintains its dependency on the `javax.xml.soap` package because the `soap` package contains the API used for creating and manipulating SOAP messages. In other words, a client sending request-response messages can use just the `javax.xml.soap` API. A Web service or client that uses one-way messaging will need to use API from both the `javax.xml.soap` and `javax.xml.messaging` packages.

Note: In this document, “JAXM 1.1_01 API” refers to the API in the `javax.xml.messaging` package; “SAAJ API” refers to the API in the `javax.xml.soap` package. “JAXM API” is a more generic term, referring to all of the API used for SOAP messaging, that is, the API in both packages.

In addition to stepping you through how to use the JAXM API, this chapter gives instructions for running the sample JAXM applications included with this tutorial as a way to help you get started. You may prefer to go through both the overview and tutorial before running the samples to make it easier to understand what the sample applications are doing, or you may prefer to explore the samples first. The overview gives some of the conceptual background behind the JAXM API to help you understand why certain things are done the way they are. The tutorial shows you how to use the basic JAXM API, giving examples and explanations of the more commonly used features. Finally, the code examples in the last part of the tutorial show how to build an application.

Overview of JAXM

This overview presents a high-level view of how JAXM messaging works and explains concepts in general terms. Its goal is to give you some terminology and a framework for the explanations and code examples that are presented in the tutorial section.

The overview looks at JAXM from three perspectives:

- Messages
- Connections
- Messaging providers

Messages

JAXM messages follow SOAP standards, which prescribe the format for messages and also specify some things that are required, optional, or not allowed. With the JAXM API, you can create XML messages that conform to the SOAP specifications simply by making Java API calls.

The Structure of an XML Document

Note: For more complete information on XML documents, see *Understanding XML* (page 177).

An XML document has a hierarchical structure with elements, subelements, sub-subelements, and so on. You will notice that many of the SAAJ classes and interfaces represent XML elements in a SOAP message and have the word *element* or *SOAP* or both in their names.

An element is also referred to as a *node*. Accordingly, the SAAJ API has the interface `Node`, which is the base class for all the classes and interfaces that represent XML elements in a SOAP message. There are also methods such as `SOAPElement.addTextNode`, `Node.detachNode`, and `Node.getValue`, which you will see how to use in the tutorial section.

What Is in a Message?

The two main types of SOAP messages are those that have attachments and those that do not.

Messages with No Attachments

The following outline and Figure 10–1 show the very high-level structure of a SOAP message with no attachments. Except for the SOAP header, all the parts listed are required.

I. SOAP message

A. SOAP part

1. SOAP envelope

a. SOAP header (optional)

b. SOAP body

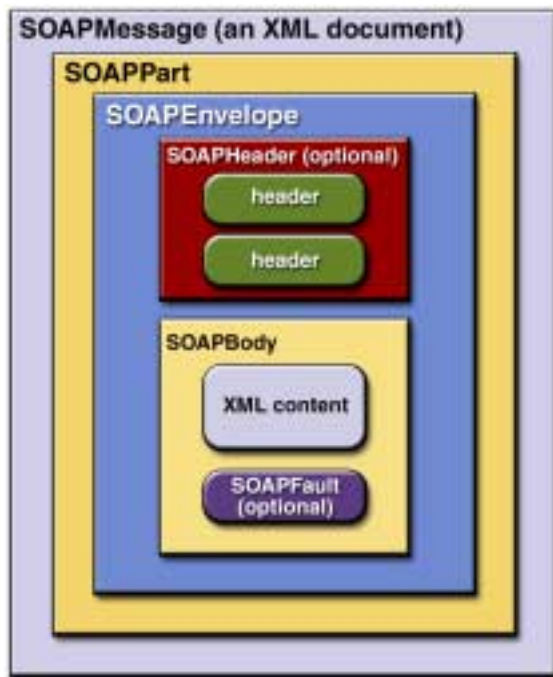


Figure 10–1 SOAPMessage Object with No Attachments

The SAAJ API provides the `SOAPMessage` class to represent a SOAP message, `SOAPPart` to represent the SOAP part, `SOAPEnvelope` to represent the SOAP envelope, and so on.

When you create a new `SOAPMessage` object, it will automatically have the parts that are required to be in a SOAP message. In other words, a new `SOAPMessage` object has a `SOAPPart` object that contains a `SOAPEnvelope` object. The `SOAPEnvelope` object in turn automatically contains an empty `SOAPHeader` object followed by an empty `SOAPBody` object. If you do not need the `SOAPHeader` object, which is optional, you can delete it. The rationale for having it automatically included is that more often than not you will need it, so it is more convenient to have it provided.

The `SOAPHeader` object may contain one or more headers with information about the sending and receiving parties and about intermediate destinations for the message. Headers may also do things such as correlate a message to previous messages, specify a level of service, and contain routing and delivery information. The `SOAPBody` object, which always follows the `SOAPHeader` object if there

is one, provides a simple way to send mandatory information intended for the ultimate recipient. If there is a `SOAPFault` object (see SOAP Faults, page 319), it must be in the `SOAPBody` object.

Messages with Attachments

A SOAP message may include one or more attachment parts in addition to the SOAP part. The SOAP part may contain only XML content; as a result, if any of the content of a message is not in XML format, it must occur in an attachment part. So, if for example, you want your message to contain an image file or plain text, your message must have an attachment part for it. Note that an attachment part can contain any kind of content, so it can contain data in XML format as well. Figure 10–2 shows the high-level structure of a SOAP message that has two attachments.

The SAAJ API provides the `AttachmentPart` class to represent the attachment part of a SOAP message. A `SOAPMessage` object automatically has a `SOAPPart` object and its required subelements, but because `AttachmentPart` objects are optional, you have to create and add them yourself. The tutorial section will walk you through creating and populating messages with and without attachment parts.

A `SOAPMessage` object may have one or more attachments. Each `AttachmentPart` object has a MIME header to indicate the type of data it contains. It may also have additional MIME headers to identify it or to give its location, which can be useful when there are multiple attachments. When a `SOAPMessage` object has one or more `AttachmentPart` objects, its `SOAPPart` object may or may not contain message content.

Another way to look at SOAP messaging is from the perspective of whether or not a messaging provider is used, which is discussed at the end of the section Messaging Providers (page 298).

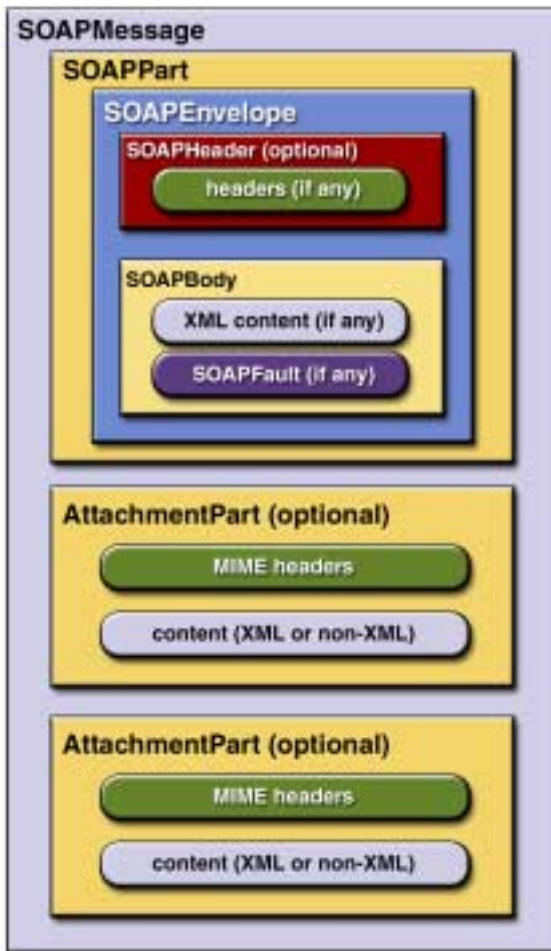


Figure 10–2 SOAPMessage Object with Two AttachmentPart Objects

Connections

All SOAP messages are sent and received over a connection. The connection can go directly to a particular destination or to a messaging provider. (A messaging provider is a service that handles the transmission and routing of messages and provides features not available when you use a connection that goes directly to its ultimate destination. Messaging providers are explained in more detail later.)

The JAXM API supplies the following class and interface to represent these two kinds of connections:

- `javax.xml.soap.SOAPConnection` — a connection from the sender directly to the receiver (a point-to-point connection)
- `javax.xml.messaging.ProviderConnection` — a connection to a messaging provider

SOAPConnection

A `SOAPConnection` object, which represents a point-to-point connection, is simple to create and use. One reason is that you do not have to do any configuration to use a `SOAPConnection` object because it does not need to run in a servlet container (like Tomcat) or in a J2EE server. It is the only kind of connection available to a client that does not use a messaging provider.

The following code fragment creates a `SOAPConnection` object and then, after creating and populating the message, uses the connection to send the message. The parameter *request* is the message being sent; *endpoint* represents where it is being sent.

```
SOAPConnectionFactory factory =
    SOAPConnectionFactory.newInstance();
SOAPConnection con = factory.createConnection();

. . . // create a request message and give it content

SOAPMessage response = con.call(request, endpoint);
```

When a `SOAPConnection` object is used, the only way to send a message is with the method `call`, which transmits its message and then blocks until it receives a reply. Because the method `call` requires that a response be returned to it, this type of messaging is referred to as *request-response* messaging.

A Web service implemented for request-response messaging must return a response to any message it receives. When the message is an update, the response is an acknowledgement that the update was received. Such an acknowledgement implies that the update was successful. Some messages may not require any response at all. The service that gets such a message is still required to send back a response because one is needed to unblock the `call` method. In this case, the response is not related to the content of the message; it is simply a message to unblock the `call` method.

Because the signature for the `javax.xml.soap.SOAPConnection.call` method changed in the SAAJ 1.1 specification, a JAXM implementation may elect not to implement the `call` method. To allow for this, there is a new exception on the `SOAPConnectionFactory` class stating that `SOAPConnection` is not implemented, which allows for a graceful failure.

Unlike a client with no messaging provider, which is limited to using only a `SOAPConnection` object, a client that uses a messaging provider is free to use a `SOAPConnection` object or a `ProviderConnection` object. It is expected that `ProviderConnection` objects will be used most of the time.

ProviderConnection

A `ProviderConnection` object represents a connection to a messaging provider. (The next section explains more about messaging providers.) When you send a message via a `ProviderConnection` object, the message goes to the messaging provider. The messaging provider forwards the message, following the message's routing instructions, until the message gets to the ultimate recipient's messaging provider, which in turn forwards the message to the ultimate recipient.

When an application is using a `ProviderConnection` object, it must use the method `ProviderConnection.send` to send a message. This method transmits the message one way and returns immediately, without having to block until it gets a response. The messaging provider that receives the message will forward it to the intended destination and return the response, if any, at a later time. The interval between sending a request and getting the response may be very short, or it may be measured in days. In this style of messaging, the original message is sent as a one-way message, and any response is sent subsequently as a one-way message. Not surprisingly, this style of messaging is referred to as *one-way* messaging.

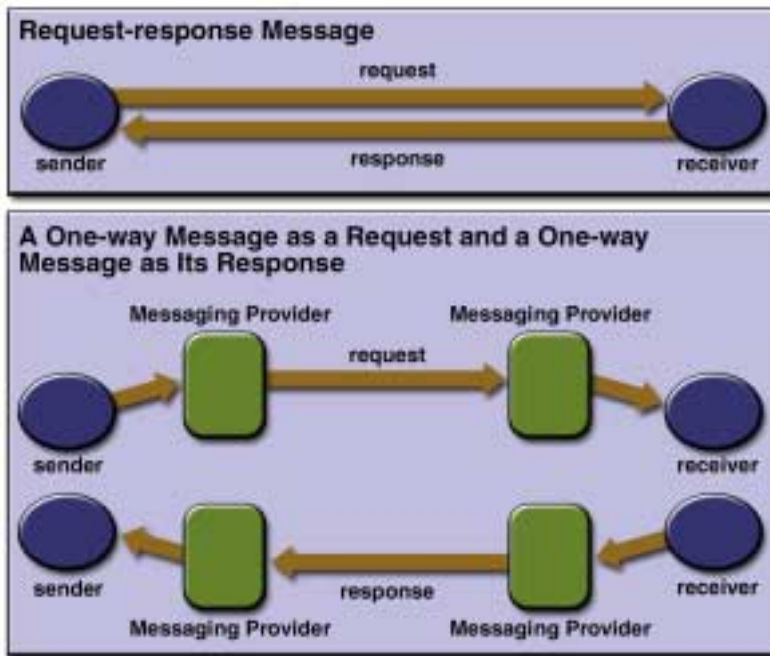


Figure 10–3 Request-response and One-way Messaging

Messaging Providers

A messaging provider is a service that handles the transmission and routing of messages. It works behind the scenes to keep track of messages and see that they are sent to the proper destination or destinations.

Transparency

One of the great features of a messaging provider is that you are not even aware of it. You just write your JAXM application, and the right things happen. For example, when you are using a messaging provider and send a message by calling the `ProviderConnection.send` method, the messaging provider receives the message and works with other parts of the communications infrastructure to perform various tasks, depending on what the message's header contains and how the messaging provider itself has been implemented. The message arrives at its final destination without your even knowing about the details involved in accomplishing the delivery.

Profiles

JAXM offers the ability to plug in additional protocols that are built on top of SOAP. A JAXM provider implementation is not required to implement features beyond what the SOAP 1.1 and SOAP with Attachments specifications require, but it is free to incorporate other standard protocols, called *profiles*, that are implemented on top of SOAP. For example, the “ebXML Message Service Specification” (available at <http://www.oasis-open.org/committees/ebxml-msg/>) defines levels of service that are not included in the two SOAP specifications. A messaging provider that is implemented to include ebXML capabilities on top of SOAP capabilities is said to support an ebXML profile. A messaging provider may support multiple profiles, but an application can use only one at a time and must have a prior agreement with each of the parties to whom it sends messages about what profile is being used.

Profiles affect a message’s headers. For example, depending on the profile, a new `SOAPMessage` object will come with certain headers already set. Also a profile implementation may provide API that makes it easier to create a header and set its content. The JAXM reference implementation includes APIs for both the ebXML and SOAP-RP profiles. The Javadoc documentation for these profiles is at `<S1STUDIO_HOME>/jwsdp/docs/jaxm/profiles/index.html`. You will find links to the Javadoc documentation for the JAXM API (the `javax.xml.soap` and `javax.xml.messaging` packages) at `<S1STUDIO_HOME>/jwsdp/docs/api/index.html`.

Note: `<S1STUDIO_HOME>` is the directory where Sun ONE Studio is installed.

Continuously Active

A messaging provider works continuously. A JAXM client may make a connection with its provider, send one or more messages, and then close the connection. The provider will store the message and then send it. Depending on how the provider has been configured, it will resend a message that was not successfully delivered until it is successfully delivered or until the limit for the number of resends is reached. Also, the provider will stay in a waiting state, ready to receive any messages that are intended for the client. The provider will store incoming messages so that when the client connects with the provider again, the provider will be able to forward the messages. In addition, the provider generates error messages as needed and maintains a log where messages and their related error messages are stored.

Intermediate Destinations

When a messaging provider is used, a message can be sent to one or more intermediate destinations before going to the final recipient. These intermediate destinations, called *actors*, are specified in the message's `SOAPHeader` object. For example, assume that a message is an incoming Purchase Order. The header might route the message to the order input desk, the order confirmation desk, the shipping desk, and the billing department. Each of these destinations is an actor that will take the appropriate action, remove the header information relevant to it, and send the message to the next actor. The default actor is the final destination, so if no actors are specified, the message is routed to the final recipient.

The attribute `actor` is used to specify an intermediate recipient. A related attribute is `mustUnderstand`, which, when its value is true, means that an actor must understand what it is supposed to do and carry it out successfully. A `SOAPHeader` object uses the method `addAttribute` to add these attributes, and the `SOAPHeaderElement` interface provides methods for setting and getting the values of these attributes.

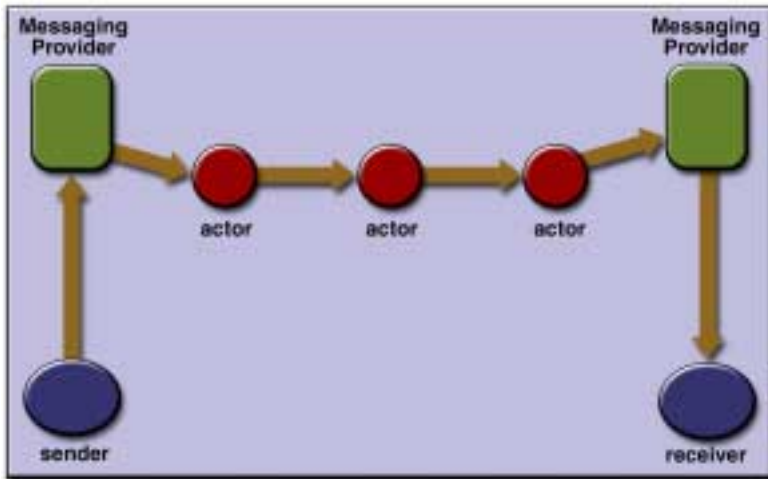


Figure 10–4 One-way Message with Intermediate Destinations

When to Use a Messaging Provider

A JAXM client may or may not use a messaging provider. Generally speaking, if you just want to be a consumer of Web services, you do not need a messaging

provider. The following list shows some of the advantages of not using a messaging provider:

- The application can be written using the J2SE platform
- The application is not required to be deployed in a servlet container or a J2EE server
- No configuration is required

The limitations of not using a messaging provider are the following:

- The client can send only request-response messages
- The client can act in the client role only

It follows that if you want to provide a Web service that is able to get and save requests that are sent to you at any time, you must use a messaging provider. You will also need to run in a container, which provides the messaging infrastructure used by the provider. A messaging provider gives you the flexibility to assume both the client and service roles, and it also lets you send one-way messages. In addition, if your messaging provider supports a protocol such as ebXML or SOAP-RP on top of SOAP, you can take advantage of the additional quality of service features that it provides.

Messaging with and without a Provider

JAXM clients can be categorized according to whether or not they use a messaging provider. Those that do not use a messaging provider can be further divided into those that run in a container and those that do not. A JAXM client that does not use a messaging provider and also does not run in a container is called a *standalone* client.

Tutorial

This section will walk you through the basics of sending a SOAP message using the JAXM API. At the end of this chapter, you will know how to do the following:

- Get a connection
- Create a message
- Add content to a message
- Send a message
- Retrieve the content from a response message
- Create and retrieve a SOAP fault element

First, we'll walk through the steps in sending a request-response message for a client that does not use a messaging provider. Then we'll do a walkthrough of a client that uses a messaging provider sending a one-way message. Both types of client may add attachments to a message, so adding attachments is covered as a separate topic. Finally, we'll see what SOAP faults are and how they work.

The section *Code Examples* (page 323) puts the code fragments you will produce into runnable applications, which you can test yourself. The JAXM part of the case study (*JAXM Distributor Service*, page 393) demonstrates how JAXM code can be used in a Web service, showing both the client and server code.

Client without a Messaging Provider

An application that does not use a messaging provider is limited to operating in a client role and can send only request-response messages. Though limited, it can make use of Web services that are implemented to do request-response messaging.

Getting a SOAPConnection Object

The first thing any JAXM client needs to do is get a connection, either a `SOAPConnection` object or a `ProviderConnection` object. The overview section (*Connections*, page 295) discusses these two types of connections and how they are used.

A client that does not use a messaging provider has only one choice for creating a connection, which is to create a `SOAPConnection` object. This kind of connec-

tion is a point-to-point connection, meaning that it goes directly from the sender to the destination (usually a URL) that the sender specifies.

The first step is to obtain a `SOAPConnectionFactory` object that you can use to create your connection. The SAAJ API makes this easy by providing the `SOAPConnectionFactory` class with a default implementation. You can get an instance of this implementation with the following line of code.

```
SOAPConnectionFactory scFactory =  
    SOAPConnectionFactory.newInstance();
```

Notice that because `newInstance` is a static method, you will always use the class name `SOAPConnectionFactory` when you invoke its `newInstance` method.

Now you can use `scFactory` to create a `SOAPConnection` object.

```
SOAPConnection con = scFactory.createConnection();
```

You will use `con` later to send the message that is created in the next part.

Creating a Message

The next step is to create a message, which you do using a `MessageFactory` object. If you are a standalone client, you can use the default implementation of the `MessageFactory` class that the SAAJ API provides. The following code fragment illustrates getting an instance of this default message factory and then using it to create a message.

```
MessageFactory factory = MessageFactory.newInstance();  
SOAPMessage message = factory.createMessage();
```

As is true of the `newInstance` method for `SOAPConnectionFactory`, the `newInstance` method for `MessageFactory` is static, so you invoke it by calling `MessageFactory.newInstance`. Note that it is possible to write your own implementation of a message factory and plug it in via system properties, but the default message factory is the one that will generally be used.

The other way to get a `MessageFactory` object is to retrieve it from a naming service where it has been registered. This way is available only to applications that use a messaging provider, and it will be covered later (in *Creating a Message*, page 311).

Parts of a Message

A `SOAPMessage` object is required to have certain elements, and the SAAJ API simplifies things for you by returning a new `SOAPMessage` object that already contains these elements. So `message`, which was created in the preceding line of code, automatically has the following:

- I. A `SOAPPart` object that contains
 - A. A `SOAPEnvelope` object that contains
 - 1. An empty `SOAPHeader` object
 - 2. An empty `SOAPBody` object

The `SOAPHeader` object, though optional, is included for convenience because most messages will use it. The `SOAPBody` object can hold the content of the message and can also contain fault messages that contain status information or details about a problem with the message. The section SOAP Faults (page 319) walks you through how to use `SOAPFault` objects.

Accessing Elements of a Message

The next step in creating a message is to access its parts so that content can be added. The `SOAPMessage` object `message`, created in the previous code fragment, is where to start. It contains a `SOAPPart` object, so you use `message` to retrieve it.

```
SOAPPart soapPart = message.getSOAPPart();
```

Next you can use `soapPart` to retrieve the `SOAPEnvelope` object that it contains.

```
SOAPEnvelope envelope = soapPart.getEnvelope();
```

You can now use `envelope` to retrieve its empty `SOAPHeader` and `SOAPBody` objects.

```
SOAPHeader header = envelope.getHeader();  
SOAPBody body = envelope.getBody();
```

Our example of a standalone client does not use a SOAP header, so you can delete it. Because all `SOAPElement` objects, including `SOAPHeader` objects, are derived from the `Node` interface, you use the method `Node.detachNode` to delete header.

```
header.detachNode();
```

Adding Content to the Body

To add content to the body, you need to create a `SOAPBodyElement` object to hold the content. When you create any new element, you also need to create an associated `Name` object to identify it. One way to create `Name` objects is by using `SOAPEnvelope` methods, so you can use `envelope` from the previous code fragment to create the `Name` object for your new element.

Note: The SAAJ API augments the `javax.xml.soap` package by adding the `SOAPFactory` class, which lets you create `Name` objects without using a `SOAPEnvelope` object. This capability is useful for creating XML elements when you are not creating an entire message. For example, JAX-RPC implementations find this ability useful. When you are not working with a `SOAPMessage` object, you do not have access to a `SOAPEnvelope` object and thus need an alternate means of creating `Name` objects. In addition to a method for creating `Name` objects, the `SOAPFactory` class provides methods for creating `Detail` objects and SOAP fragments. You will find an explanation of `Detail` objects in the SOAP Fault sections Overview (page 319) and Creating and Populating a `SOAPFault` Object (page 321).

`Name` objects associated with `SOAPBody` and `SOAPHeader` objects must be fully qualified; that is, they must be created with a local name, a prefix for the namespace being used, and a URI for the namespace. Specifying a namespace for an element makes clear which one is meant if there is more than one element with the same local name.

The code fragment that follows retrieves the `SOAPBody` object `body` from `envelope`, creates a `Name` object for the element to be added, and adds a new `SOAPBodyElement` object to `body`.

```
SOAPBody body = envelope.getBody();
Name bodyName = envelope.createName("GetLastTradePrice",
    "m", "http://wombat.ztrade.com");
SOAPBodyElement gltp = body.addBodyElement(bodyName);
```

At this point, `body` contains a `SOAPBodyElement` object identified by the `Name` object `bodyName`, but there is still no content in `gltp`. Assuming that you want to get a quote for the stock of Sun Microsystems, Inc., you need to create a child element for the symbol using the method `addChildElement`. Then you need to give it the stock symbol using the method `addTextNode`. The `Name` object for the

new `SOAPElement` object `symbol` is initialized with only a local name, which is allowed for child elements.

```
Name name = envelope.createName("symbol");
SOAPElement symbol = gltp.addChildElement(name);
symbol.addTextNode("SUNW");
```

You might recall that the headers and content in a `SOAPPart` object must be in XML format. The JAXM API takes care of this for you, building the appropriate XML constructs automatically when you call methods such as `addBodyElement`, `addChildElement`, and `addTextNode`. Note that you can call the method `addTextNode` only on an element such as `bodyElement` or any child elements that are added to it. You cannot call `addTextNode` on a `SOAPHeader` or `SOAPBody` object because they contain elements, not text.

The content that you have just added to your `SOAPBody` object will look like the following when it is sent over the wire:

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  <SOAP-ENV:Body>
    <m:GetLastTradePrice xmlns:m="http://wombat.ztrade.com">
      <symbol>SUNW</symbol>
    </m:GetLastTradePrice>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Let's examine this XML excerpt line by line to see how it relates to your JAXM code. Note that an XML parser does not care about indentations, but they are generally used to indicate element levels and thereby make it easier for a human reader to understand.

JAXM code:

```
SOAPPart soapPart = message.getSOAPPart();
SOAPEnvelope envelope = soapPart.getEnvelope();
```

XML it produces:

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  . . . . . (intervening elements omitted)
</SOAP-ENV:Envelope>
```

The outermost element in this XML example is the SOAP envelope element, indicated by SOAP-ENV:Envelope. Envelope is the name of the element, and SOAP-ENV is the namespace prefix. The interface SOAPEnvelope represents a SOAP envelope.

The first line signals the beginning of the SOAP envelope element, and the last line signals the end of it; everything in between is part of the SOAP envelope. The second line has an attribute for the SOAP envelope element. xmlns stands for “XML namespace,” and its value is the URI of the namespace associated with Envelope. This attribute is automatically included for you.

JAXM code:

```
SOAPBody body = envelope.getBody();
```

XML it produces:

```
<SOAP-ENV:Body>
. . . . .
</SOAP-ENV:Body>
```

These two lines mark the beginning and end of the SOAP body, represented in JAXM by a SOAPBody object.

JAXM code:

```
Name bodyName = envelope.createName("GetLastTradePrice",
    "m", "http://wombat.ztrade.com");
SOAPBodyElement gltp = body.addBodyElement(bodyName);
```

XML it produces:

```
<m:GetLastTradePrice xmlns:m="http://wombat.ztrade.com">
. . . . .
</m:GetLastTradePrice>
```

These lines are what the SOAPBodyElement gltp in your code represents. "GetLastTradePrice" is its local name, "m" is its namespace prefix, and "http://wombat.ztrade.com" is its namespace URI.

JAXM code:

```
Name name = envelope.createName("symbol");
SOAPElement symbol = gltp.addChildElement(name);
symbol.addTextNode("SUNW");
```

XML it produces:

```
<symbol>SUNW</symbol>
```

The String "SUNW" is the message content that your recipient, the stock quote service, receives.

Sending a Message

A standalone client uses a `SOAPConnection` object and must therefore use the `SOAPConnection` method `call` to send a message. This method takes two arguments, the message being sent and the destination to which the message should go. This message is going to the stock quote service indicated by the URL object endpoint.

```
java.net.URL endpoint = new URL(
    "http://wombat.ztrade.com/quotes");

SOAPMessage response = con.call(message, endpoint);
```

Your message sent the stock symbol SUNW; the `SOAPMessage` object response should contain the last stock price for Sun Microsystems, which you will retrieve in the next section.

A connection uses a fair amount of resources, so it is a good idea to close a connection as soon as you are through using it.

```
con.close();
```

Getting the Content of a Message

The initial steps for retrieving a message's content are the same as those for giving content to a message: You first access the `SOAPBody` object, using the message to get the envelope and the envelope to get the body. Then you access its `SOAPBodyElement` object because that is the element to which content was added in the example. (In a later section you will see how to add content directly to the `SOAPBody` object, in which case you would not need to access the `SOAPBodyElement` object for adding content or for retrieving it.) To get the content, which was added with the method `SOAPElement.addTextNode`, you call the method `Node.getValue`. Note that `getValue` returns the value of the immediate child of the element that calls the method. Therefore, in the following code fragment, the

method `getValue` is called on `bodyElement`, the element on which the method `addTextNode` was called.

In order to access `bodyElement`, you need to call the method `getChildElement` on `body`. Passing `bodyName` to `getChildElement` returns a `java.util.Iterator` object that contains all of the child elements identified by the `Name` object `bodyName`. You already know that there is only one, so just calling the method `next` on it will return the `SOAPBodyElement` you want. Note that the method `Iterator.next` returns a `Java Object`, so it is necessary to cast the `Object` it returns to a `SOAPBodyElement` object before assigning it to the variable `bodyElement`.

```
SOAPPart sp = response.getSOAPPart();
SOAPEnvelope env = sp.getEnvelope();
SOAPBody sb = env.getBody();
java.util.Iterator it = sb.getChildElements(bodyName);
SOAPBodyElement bodyElement = (SOAPBodyElement)it.next();
String lastPrice = bodyElement.getValue();
System.out.print("The last price for SUNW is ");
System.out.println(lastPrice);
```

If there were more than one element with the name `bodyName`, you would have had to use a `while` loop using the method `Iterator.hasNext` to make sure that you got all of them.

```
while (it.hasNext()) {
    SOAPBodyElement bodyElement = (SOAPBodyElement)it.next();
    String lastPrice = bodyElement.getValue();
    System.out.print("The last price for SUNW is ");
    System.out.println(lastPrice);
}
```

At this point, you have seen how to send a request-response message as a standalone client. You have also seen how to get the content from the response. The next part shows you how to send a message using a messaging provider.

Client with a Messaging Provider

Using a messaging provider gives you more flexibility than a standalone client has because it can take advantage of the additional functionality that a messaging provider can offer.

Getting a ProviderConnection Object

Whereas a `SOAPConnection` object is a point-to-point connection directly to a particular URL, a `ProviderConnection` object is a connection to a messaging provider. With this kind of connection, all messages that you send or receive go through the messaging provider.

As with getting a `SOAPConnection` object, the first step is to get a connection factory, but in this case, it is a `ProviderConnectionFactory` object. You can obtain a `ProviderConnectionFactory` object by retrieving it from a naming service. This is possible when your application is using a messaging provider and is deployed in a servlet container or J2EE server. With a `ProviderConnectionFactory` object, you can create a connection to a particular messaging provider and thus be able to use the capabilities of a profile that the messaging provider supports.

To get a `ProviderConnectionFactory` object, you first supply the logical name of your messaging provider to the container at deployment time. This is the name associated with your messaging provider that has been registered with a naming service based on the Java Naming and Directory Interface™ (JNDI) API. You can then do a lookup using this name to obtain a `ProviderConnectionFactory` object that will create connections to your messaging provider. For example, if the name registered for your messaging provider is “ProviderABC”, you can do a lookup on “ProviderABC” to get a `ProviderConnectionFactory` object and use it to create a connection to your messaging provider. This is what is done in the following code fragment. The first two lines use methods from the JNDI API to retrieve the `ProviderConnectionFactory` object, and the last line uses a method from the JAXM API to create the connection to the messaging provider. Note that because the JNDI method `lookup` returns a `Java Object`, you must convert it to a `ProviderConnectionFactory` object before assigning it to the variable `pcFactory`.

```
Context ctx = new InitialContext();
ProviderConnectionFactory pcFactory =
    (ProviderConnectionFactory)ctx.lookup("ProviderABC");

ProviderConnection pcCon = pcFactory.createConnection();
```

You will use `pcCon`, which represents a connection to your messaging provider, to get information about your messaging provider and to send the message you will create in the next section.

Creating a Message

You create all JAXM messages by getting a `MessageFactory` object and using it to create the `SOAPMessage` object. For the standalone client example, you simply used the default `MessageFactory` object obtained via the method `MessageFactory.newInstance`. However, when you are using a messaging provider, you obtain the `MessageFactory` object in a different way.

Getting a MessageFactory

If you are using a messaging provider, you create a `MessageFactory` object by using the method `ProviderConnection.createMessageFactory`. In addition, you pass it a `String` indicating the profile you want to use. To find out which profiles your messaging provider supports, you need to get a `ProviderMetaData` object with information about your provider. This is done by calling the method `getMetaData` on the connection to your provider. Then you need to call the method `getSupportedProfiles` to get an array of the profiles your messaging provider supports. Supposing that you want to use the ebXML profile, you need to see if any of the profiles in the array matches "ebxml". If there is a match, that profile is assigned to the variable `profile`, which can then be passed to the method `createMessageFactory`.

```
ProviderMetaData metaData = pcCon.getMetaData();
String[] supportedProfiles = metaData.getSupportedProfiles();
String profile = null;

for (int i=0; i < supportedProfiles.length; i++) {
    if (supportedProfiles[i].equals("ebxml")) {
        profile = supportedProfiles[i];
        break;
    }
}

MessageFactory factory = pcCon.createMessageFactory(profile);
```

You can now use `factory` to create a `SOAPMessage` object that conforms to the ebXML profile. This example uses the minimal ebXML profile implementation included in the Java WSDP. Note that the following line of code uses the class `EbXMLMessageImpl`, which is defined in the ebXML profile implementation and is not part of the JAXM API.

```
EbXMLMessageImpl message =
    (EbXMLMessageImpl) factory.createMessage();
```

For this profile, instead of using Endpoint objects, you indicate Party objects for the sender and the receiver. This information will appear in the message's header, and the messaging provider will use it to determine where to send the message. The following lines of code use the methods `setSender` and `setReceiver`, which are defined in the `EbXMLMessageImpl` implementation. These methods not only create a `SOAPHeader` object but also give it content. You can use these methods because your `SOAPMessage` object is an `EbXMLMessageImpl` object, giving you access to the methods defined in `EbXMLMessageImpl`.

```
message.setSender(new Party("http://grand.products.com"));
message.setReceiver(new Party("http://whiz.gizmos.com"));
```

You can view the Javadoc comments for the ebXML and SOAP-RP profile implementations provided in Sun ONE Studio at the following location:

```
<S1STUDIO_HOME>/jwsdp/docs/jaxm/profile/com/sun/xml/messaging/
```

If you are not using a profile or you want to set content for a header not covered by your profile's implementation, you need to follow the steps shown in the next section.

Adding Content to the Header

To add content to the header, you need to create a `SOAPHeaderElement` object. As with all new elements, it must have an associated `Name` object, which you create using the message's `SOAPEnvelope` object.

The following code fragment retrieves the `SOAPHeader` object from `envelope` and adds a new `SOAPHeaderElement` object to it.

```
SOAPHeader header = envelope.getHeader();
Name headerName = envelope.createName("Purchase Order",
    "PO", "http://www.sonata.com/order");
SOAPHeaderElement headerElement =
    header.addHeaderElement(headerName);
```

At this point, `header` contains the `SOAPHeaderElement` object `headerElement` identified by the `Name` object `headerName`. Note that the `addHeaderElement` method both creates `headerElement` and adds it to `header`.

Now that you have identified `headerElement` with `headerName` and added it to `header`, the next step is to add content to `headerElement`, which the next line of code does with the method `addTextNode`.

```
headerElement.addTextNode("order");
```

Now you have the `SOAPHeader` object `header` that contains a `SOAPHeaderElement` object whose content is "order".

Adding Content to the SOAP Body

The process for adding content to the `SOAPBody` object is the same for clients using a messaging provider as it is for standalone clients. This is also the same as the process for adding content to the `SOAPHeader` object. You access the `SOAPBody` object, add a `SOAPBodyElement` object to it, and add text to the `SOAPBodyElement` object. It is possible to add additional `SOAPBodyElement` objects, and it is possible to add subelements to the `SOAPBodyElement` objects with the method `addChildElement`. For each element or child element, you add content with the method `addTextNode`.

The section on the standalone client demonstrated adding one `SOAPBodyElement` object, adding a child element, and giving it some text. The following example shows adding more than one `SOAPBodyElement` and adding text to each of them.

The code first creates the `SOAPBodyElement` object `purchaseLineItems`, which has a fully-qualified namespace associated with it. That is, the `Name` object for it has a local name, a namespace prefix, and a namespace URI. As you saw earlier, a `SOAPBodyElement` object is required to have a fully-qualified namespace, but child elements added to it may have `Name` objects with only the local name.

```
SOAPBody body = envelope.getBody();
Name bodyName = envelope.createName("PurchaseLineItems", "PO",
    "http://sonata.fruitsgalore.com");
SOAPBodyElement purchaseLineItems =
    body.addBodyElement(bodyName);

Name childName = envelope.createName("Order");
SOAPElement order =
    purchaseLineItems.addChildElement(childName);

childName = envelope.createName("Product");
SOAPElement product = order.addChildElement(childName);
product.addTextNode("Apple");

childName = envelope.createName("Price");
```

```

SOAPElement price = order.addChildElement(childName);
price.addTextNode("1.56");

childName = envelope.createName("Order");
SOAPElement order2 =
    purchaseLineItems.addChildElement(childName);

childName = envelope.createName("Product");
SOAPElement product2 = order2.addChildElement(childName);
product2.addTextNode("Peach");

childName = envelope.createName("Price");
SOAPElement price2 = order2.addChildElement(childName);
price2.addTextNode("1.48");

```

The JAXM code in the preceding example produces the following XML in the SOAP body:

```

<P0:PurchaseLineItems
  xmlns:P0="http://www.sonata.fruitsgalore/order">
  <Order>
    <Product>Apple</Product>
    <Price>1.56</Price>
  </Order>

  <Order>
    <Product>Peach</Product>
    <Price>1.48</Price>
  </Order>
</P0:PurchaseLineItems>

```

Adding Content to the SOAPPart Object

If the content you want to send is in a file, JAXM provides an easy way to add it directly to the SOAPPart object. This means that you do not access the SOAPBody object and build the XML content yourself, as you did in the previous section.

To add a file directly to the SOAPPart object, you use a `javax.xml.transform.Source` object from JAXP (the Java API for XML Processing). There are three types of Source objects: `SAXSource`, `DOMSource`, and `StreamSource`. A `StreamSource` object holds content as an XML document. `SAXSource` and `DOMSource` objects hold content along with the instructions for transforming the content into an XML document.

The following code fragment uses JAXP API to build a `DOMSource` object that is passed to the `SOAPPart.setContent` method. The first two lines of code get a

DocumentBuilderFactory object and use it to create the DocumentBuilder object builder. Then builder parses the content file to produce a Document object, which is used to initialize a new DOMSource object.

```
DocumentBuilderFactory dbFactory =
    DocumentBuilderFactory.newInstance();
DocumentBuilder builder = dbFactory.newDocumentBuilder();
Document doc = builder.parse("file:///music/order/soap.xml");
DOMSource domSource = new DOMSource(doc);
```

The following two lines of code access the SOAPPart object (using the SOAPMessage object message) and set the new DOMSource object as its content. The method SOAPPart.setContent not only sets content for the SOAPBody object but also sets the appropriate header for the SOAPHeader object.

```
SOAPPart soapPart = message.getSOAPPart();
soapPart.setContent(domSource);
```

You will see other ways to add content to a message in the section on AttachmentPart objects. One big difference to keep in mind is that a SOAPPart object must contain only XML data, whereas an AttachmentPart object may contain any type of content.

Sending the Message

When the connection is a ProviderConnection object, messages have to be sent using the method ProviderConnection.send. This method sends the message passed to it and returns immediately. Unlike the SOAPConnection method call, it does not have to block until it receives a response, which leaves the application free to do other things.

The send method takes only one argument, the message to be sent. It does not need to be given the destination because the messaging provider can use information in the header to figure out where the message needs to go.

```
pcCon.send(message);
pcCon.close();
```

Adding Attachments

Adding AttachmentPart objects to a message is the same for all clients, whether they use a messaging provider or not. As noted in earlier sections, you

can put any type of content, including XML, in an `AttachmentPart` object. And because the SOAP part can contain only XML content, you must use an `AttachmentPart` object for any content that is not in XML format.

Creating an AttachmentPart Object and Adding Content

The `SOAPMessage` object creates an `AttachmentPart` object, and the message also has to add the attachment to itself after content has been added. The `SOAPMessage` class has three methods for creating an `AttachmentPart` object.

The first method creates an attachment with no content. In this case, an `AttachmentPart` method is used later to add content to the attachment.

```
AttachmentPart attachment = message.createAttachmentPart();
```

You add content to attachment with the `AttachmentPart` method `setContent`. This method takes two parameters, a Java Object for the content, and a String object that gives the content type. Content in the `SOAPBody` part of a message automatically has a `Content-Type` header with the value "text/xml" because the content has to be in XML. In contrast, the type of content in an `AttachmentPart` object has to be specified because it can be any type.

Each `AttachmentPart` object has one or more headers associated with it. When you specify a type to the method `setContent`, that type is used for the header `Content-Type`. `Content-Type` is the only header that is required. You may set other optional headers, such as `Content-Id` and `Content-Location`. For convenience, JAXM provides `get` and `set` methods for the headers `Content-Type`, `Content-Id`, and `Content-Location`. These headers can be helpful in accessing a particular attachment when a message has multiple attachments. For example, to access the attachments that have particular headers, you call the `SOAPMessage` method `getAttachments` and pass it the header or headers you are interested in.

The following code fragment shows one of the ways to use the method `setContent`. The Java Object being added is a `String`, which is plain text, so the second argument has to be "text/plain". The code also sets a content identifier, which can be used to identify this `AttachmentPart` object. After you have added

content to attachment, you need to add attachment to the SOAPMessage object, which is done in the last line.

```
String stringContent = "Update address for Sunny Skies " +
    "Inc., to 10 Upbeat Street, Pleasant Grove, CA 95439";

attachment.setContent(stringContent, "text/plain");
attachment.setContentId("update_address");

message.addAttachmentPart(attachment);
```

The variable `attachment` now represents an `AttachmentPart` object that contains the `String` `stringContent` and has a header that contains the `String` “text/plain”. It also has a `Content-Id` header with “update_address” as its value. And now `attachment` is part of `message`.

Let’s say you also want to attach a JPEG image showing how beautiful the new location is. In this case, the second argument passed to `setContent` must be “image/jpeg” to match the content being added. The code for adding an image might look like the following. For the first attachment, the `Object` passed to the method `setContent` was a `String`. In this case, it is a stream.

```
AttachmentPart attachment2 = message.createAttachmentPart();

byte[] jpegData = . . . ;
ByteArrayInputStream stream =
    new ByteArrayInputStream(jpegData);

attachment2.setContent(stream, "image/jpeg");

message.addAttachmentPart(attachment);
```

The other two `SOAPMessage.createAttachment` methods create an `AttachmentPart` object complete with content. One is very similar to the `AttachmentPart.setContent` method in that it takes the same parameters and does essentially the same thing. It takes a `Java Object` containing the content and a `String` giving the content type. As with `AttachmentPart.setContent`, the `Object` may be a `String`, a stream, a `javax.xml.transform.Source` object, or a `javax.activation.DataHandler` object. You have already seen an example of using a `Source` object as content. The next example will show how to use a `DataHandler` object for content.

The other method for creating an `AttachmentPart` object with content takes a `DataHandler` object, which is part of the JavaBeans™ Activation Framework (JAF). Using a `DataHandler` object is fairly straightforward. First you create a

`java.net.URL` object for the file you want to add as content. Then you create a `DataHandler` object initialized with the `URL` object and pass it to the method `createAttachmentPart`.

```
URL url = new URL("http://greatproducts.com/gizmos/img.jpg");
DataHandler dh = new DataHandler(url);
AttachmentPart attachment = message.createAttachmentPart(dh);
attachment.setContentId("gyro_image");

message.addAttachmentPart(attachment);
```

You might note two things about the previous code fragment. First, it sets a header for Content-ID with the method `setContentId`. This method takes a `String` that can be whatever you like to identify the attachment. Second, unlike the other methods for setting content, this one does not take a `String` for Content-Type. This method takes care of setting the Content-Type header for you, which is possible because one of the things a `DataHandler` object does is determine the data type of the file it contains.

Accessing an AttachmentPart Object

If you receive a message with attachments or want to change an attachment to a message you are building, you will need to access the attachment. When it is given no argument, the method `SOAPMessage.getAttachments` returns a `java.util.Iterator` object over all the `AttachmentPart` objects in a message. The following code prints out the content of each `AttachmentPart` object in the `SOAPMessage` object `message`.

```
java.util.Iterator it = message.getAttachments();
while (it.hasNext()) {
    AttachmentPart attachment = (AttachmentPart)it.next();
    Object content = attachment.getContent();
    String id = attachment.getContentId();
    System.out.print("Attachment " + id + " contains: " +
        content);
    System.out.println("");
}
```

Summary

In this section, you have been introduced to the basic JAXM API. You have seen how to create and send SOAP messages as a standalone client and as a client using a messaging provider. You have walked through adding content to a SOAP

header and a SOAP body and also walked through creating attachments and giving them content. In addition, you have seen how to retrieve the content from the SOAP part and from attachments. In other words, you have walked through using the basic JAXM API.

SOAP Faults

This section expands on the basic JAXM API by showing you how to use the API for creating and accessing a SOAP Fault element in an XML message.

Overview

If you send a message that was not successful for some reason, you may get back a response containing a SOAP Fault element that gives you status information, error information, or both. There can be only one SOAP Fault element in a message, and it must be an entry in the SOAP Body. The SOAP 1.1 specification defines only one Body entry, which is the SOAP Fault element. Of course, the SOAP Body may contain other Body entries, but the SOAP Fault element is the only one that has been defined.

A `SOAPFault` object, the representation of a SOAP Fault element in the JAXM API, is similar to an `Exception` object in that it conveys information about a problem. However, a `SOAPFault` object is quite different in that it is an element in a message's `SOAPBody` object rather than part of the try/catch mechanism used for `Exception` objects. Also, as part of the `SOAPBody` object, which provides a simple means for sending mandatory information intended for the ultimate recipient, a `SOAPFault` object only reports status or error information. It does not halt the execution of an application the way an `Exception` object can.

Various parties may supply a `SOAPFault` object in a message. If you are a standalone client using the SAAJ API, and thus sending point-to-point messages, the recipient of your message may add a `SOAPFault` object to the response to alert you to a problem. For example, if you sent an order with an incomplete address for where to send the order, the service receiving the order might put a `SOAPFault` object in the return message telling you that part of the address was missing.

In another scenario, if you use the JAXM 1.1_01 API (or later) in order to use a messaging provider, the messaging provider may be the one to supply a `SOAPFault` object. For example, if the provider has not been able to deliver a message because a server is unavailable, the provider might send you a message with a

SOAPFault object containing that information. In this case, there was nothing wrong with the message itself, so you can try sending it again later without any changes. In the previous example, however, you would need to add the missing information before sending the message again.

A SOAPFault object contains the following elements:

- A **fault code** — always required

The SOAP 1.1 specification defines a set of fault code values in section 4.4.1, which a developer may extend to cover other problems. The default fault codes defined in the specification relate to the JAXM API as follows:

- **VersionMismatch** — the namespace for a SOAPEnvelope object was invalid
- **MustUnderstand** — an immediate child element of a SOAPHeader object had its mustUnderstand attribute set to "1", and the processing party did not understand the element or did not obey it
- **Client** — the SOAPMessage object was not formed correctly or did not contain the information needed to succeed
- **Server** — the SOAPMessage object could not be processed because of a processing error, not because of a problem with the message itself
- A **fault string** — always required

A human readable explanation of the fault

- A **fault actor** — required if the SOAPHeader object contains one or more actor attributes; optional if no actors are specified, meaning that the only actor is the ultimate destination

The fault actor, which is specified as a URI, identifies who caused the fault. For an explanation of what an actor is, see the section Intermediate Destinations (page 300).

- A **Detail object** — required if the fault is an error related to the SOAPBody object

If, for example, the fault code is "Client", indicating that the message could not be processed because of a problem in the SOAPBody object, the SOAPFault object must contain a Detail object that gives details about the problem. If a SOAPFault object does not contain a Detail object, it can be assumed that the SOAPBody object was processed successfully.

Creating and Populating a SOAPFault Object

You have already seen how to add content to a SOAPBody object; this section will walk you through adding a SOAPFault object to a SOAPBody object and then adding its constituent parts.

As with adding content, the first step is to access the SOAPBody object.

```
SOAPEnvelope envelope =  
    msg.getSOAPPart().getEnvelope();  
SOAPBody body = envelope.getBody();
```

With the SOAPBody object body in hand, you can use it to create a SOAPFault object with the following line of code.

```
SOAPFault fault = body.addFault();
```

The following code uses convenience methods to add elements and their values to the SOAPFault object fault. For example, the method setFaultCode creates an element, adds it to fault, and adds a Text node with the value "Server".

```
fault.setFaultCode("Server");  
fault.setFaultActor("http://gizmos.com/orders");  
fault.setFaultString("Server not responding");
```

The SOAPFault object fault created in the previous lines of code indicates that the cause of the problem is an unavailable server and that the actor at "http://gizmos.com/orders" is having the problem. If the message were being routed only to its ultimate destination, there would have been no need for setting a fault actor. Also note that fault does not have a Detail object because it does not relate to the SOAPBody object.

The following code fragment creates a SOAPFault object that includes a Detail object. Note that a SOAPFault object may have only one Detail object, which is simply a container for DetailEntry objects, but the Detail object may have multiple DetailEntry objects. The Detail object in the following lines of code has two DetailEntry objects added to it.

```
SOAPFault fault = body.addFault();  
  
fault.setFaultCode("Client");  
fault.setFaultString("Message does not have necessary info");  
  
Detail detail = fault.addDetail();
```

```

Name entryName = envelope.createName("order", "PO",
    "http://gizmos.com/orders/");
DetailEntry entry = detail.addDetailEntry(entryName);
entry.addTextNode("quantity element does not have a value");

Name entryName2 = envelope.createName("confirmation", "PO",
    "http://gizmos.com/confirm");
DetailEntry entry2 = detail.addDetailEntry(entryName2);
entry2.addTextNode("Incomplete address: no zip code");

```

Retrieving Fault Information

Just as the `SOAPFault` interface provides convenience methods for adding information, it also provides convenience methods for retrieving that information. The following code fragment shows what you might write to retrieve fault information from a message you received. In the code fragment, `newmsg` is the `SOAPMessage` object that has been sent to you. Because a `SOAPFault` object must be part of the `SOAPBody` object, the first step is to access the `SOAPBody` object. Then the code tests to see if the `SOAPBody` object contains a `SOAPFault` object. If so, the code retrieves the `SOAPFault` object and uses it to retrieve its contents. The convenience methods `getFaultCode`, `getFaultString`, and `getFaultActor` make retrieving the values very easy.

```

SOAPBody body =
    newmsg.getSOAPPart().getEnvelope().getBody();
if ( body.hasFault() ) {
    SOAPFault newFault = body.getFault();
    String code = newFault.getFaultCode();
    String string = newFault.getFaultString();
    String actor = newFault.getFaultActor();

```

Next the code prints out the values it just retrieved. Not all messages are required to have a fault actor, so the code tests to see if there is one. Testing whether the variable `actor` is `null` works because the method `getFaultActor` returns `null` if a fault actor has not been set.

```

    System.out.println("SOAP fault contains: ");
    System.out.println("    fault code = " + code);
    System.out.println("    fault string = " + string);

    if ( actor != null ) {
        System.out.println("    fault actor = " + actor);
    }
}

```

The final task is to retrieve the `Detail` object and get its `DetailEntry` objects. The code uses the `SOAPFault` object `newFault` to retrieve the `Detail` object `newDetail`, and then it uses `newDetail` to call the method `getDetailEntries`. This method returns the `java.util.Iterator` object `it`, which contains all of the `DetailEntry` objects in `newDetail`. Not all `SOAPFault` objects are required to have a `Detail` object, so the code tests to see whether `newDetail` is `null`. If it is not, the code prints out the values of the `DetailEntry` object(s) as long as there are any.

```
Detail newDetail = newFault.getDetail();
if ( newDetail != null) {
    Iterator it = newDetail.getDetailEntries();
    while ( it.hasNext() ) {
        DetailEntry entry = (DetailEntry)it.next();
        String value = entry.getValue();
        System.out.println("  Detail entry = " + value);
    }
}
```

In summary, you have seen how to add a `SOAPFault` object and its contents to a message as well as how to retrieve the information in a `SOAPFault` object. A `SOAPFault` object, which is optional, is added to the `SOAPBody` object to convey status or error information. It must always have a fault code and a `String` explanation of the fault. A `SOAPFault` object must indicate the actor that is the source of the fault only when there are multiple actors; otherwise, it is optional. Similarly, the `SOAPFault` object must contain a `Detail` object with one or more `DetailEntry` objects only when the contents of the `SOAPBody` object could not be processed successfully.

Code Examples

The first part of this tutorial used code fragments to walk you through the fundamentals of using the JAXM API. In this section, you will use some of those code fragments to create applications. First, you will see the program `Request.java`. Then you will see how to create and run the following applications:

- `UddiPing.java` — A simple standalone example that sends a message to a UDDI test registry and receives a response
- `SOAPFaultTest.java` — A simple standalone example that shows how to specify SOAP faults

- JAXM Simple — A simple example of sending and receiving a message using the local provider
- SAAJ Simple — An application similar to the Simple example except that it is written using only the SAAJ API
- JAXM Translator — An application that uses a simple translation service to translate a given word into different languages
- JAXM Tags — An example that uses JavaServer Pages tags to generate and consume a SOAP message
- JAXM Provider — A simple example of a JAXM provider
- JAXM Provider Administrator — A simple web-based administrative tool for the JAXM provider

This list presents the sample applications according to what they do. You can also look at the sample applications as examples of the three possible types of JAXM client:

- **Those that do not use a messaging provider and also do not run in a container**

These are called *standalone* applications. The samples `UddiPing` and `SOAPFaultTest` are examples of standalone clients.

- **Those that do not use a messaging provider and run in a container**
The samples `JAXM Simple`, `SAAJ Simple`, `JAXM Translator`, and `JAXM Tags` are examples of this type.

- **Those that use a messaging provider and run in a container**

There are no samples of this type at this release.

Setting the Classpath

Before you can compile and run the examples, you need to set the classpath for both compiling and executing the programs.

To set the compilation and execution classpaths:

1. Choose Options from the Tools menu.
2. Expand the Building node, then the Compiler Types node.
3. Choose External Compilation.
4. Select the Expert tab.
5. Click the Class Path property and open the property editor.

6. Click Add JAR/Zip.
7. In the file chooser, navigate to the directory `<S1STUDIO_HOME>/jwsdp/common/lib` and choose the file `jaxm-api.jar`.
8. Click OK.
9. Click Add JAR/Zip again and repeat steps 7-8. This time, choose the file `commons-logging.jar`.
10. Expand the Debugging and Executing node, then the Execution Types node.
11. Choose External Execution.
12. Select the Expert tab.
13. Click the Class Path property, then double-click the ellipsis in the value field.
14. In the property editor, click Add JAR/Zip.
15. In the file chooser, navigate to the directory `<S1STUDIO_HOME>/jwsdp/common/lib` and choose the `jaxm-api.jar` file.
16. Click OK.
17. Click Close in the Options window.

Changing Server Permissions

Two of the examples, JAXM Translator and JAXM Tags, require certain permissions to be granted in order to run successfully. To make these changes, perform the following steps:

1. Open the following file in an editor:

```
<S1AS7_HOME>/domains/domain1/server1/config/server.policy
```

2. Find the following line:

```
permission java.util.PropertyPermission "*", "read";
```

3. Add write permission by changing the line as follows:

```
permission java.util.PropertyPermission "*", "read,write";
```

4. Append the following lines of text to the end of the file. The entire string between `grant codebase` and the opening curly brace (`{`) should be on one line:

```
// This permission applies to the privileged jaxm-tags webapp
grant codeBase "file:${com.sun.aas.installRoot}/domains/
domain1/server1/applications/j2ee-modules/-" {
permission java.util.logging.LoggingPermission "control", "";
};
```

5. Restart the server instance so the changes will take effect.

The Request.java Program

The class `Request.java` puts together the code fragments used in the section `Client without a Messaging Provider` (page 302) and adds what is needed to make it a complete example of a client sending a request-response message. In addition to putting all the code together, it adds `import` statements, a `main` method, and a `try/catch` block with exception handling. The file `Request.java`, shown here in its entirety, is a standalone client application that uses the SAAJ API (the `javax.xml.soap` package). It does not need to use the `javax.xml.messaging` package because it does not use a messaging provider.

```
import javax.xml.soap.*;
import java.util.*;
import java.net.URL;
public class Request {
    public static void main(String[] args){
        try {
            SOAPConnectionFactory scFactory =
                SOAPConnectionFactory.newInstance();
            SOAPConnection con = scFactory.createConnection();

            MessageFactory factory =
                MessageFactory.newInstance();
            SOAPMessage message = factory.createMessage();

            SOAPPart soapPart = message.getSOAPPart();
            SOAPEnvelope envelope = soapPart.getEnvelope();
            SOAPHeader header = envelope.getHeader();
            SOAPBody body = envelope.getBody();
            header.detachNode();

            Name bodyName = envelope.createName(
```

```

        "GetLastTradePrice", "m",
        "http://wombats.ztrade.com");
    SOAPBodyElement gltp =
        body.addBodyElement(bodyName);

    Name name = envelope.createName("symbol");
    SOAPElement symbol = gltp.addChildElement(name);
    symbol.addTextNode("SUNW");

    URL endpoint = new URL
        ("http://wombat.ztrade.com/quotes");
    SOAPMessage response =
        con.call(message, endpoint);

    con.close();

    SOAPPart sp = response.getSOAPPart();
    SOAPEnvelope se = sp.getEnvelope();
    SOAPBody sb = se.getBody();

    Iterator it = sb.getChildElements(bodyName);
    SOAPBodyElement bodyElement =
        (SOAPBodyElement)it.next();
    String lastPrice = bodyElement.getValue();

    System.out.print("The last price for SUNW is ");
    System.out.println(lastPrice);

    } catch (Exception ex) {
        ex.printStackTrace();
    }
}

```

In order for `Request.java` to be runnable, the second argument supplied to the method `call` has to be a valid existing URI, which is not true in this case. See the JAXM code in the case study for similar code that you can run (JAXM Client, page 403). Also, the application in the next section is one that you can run.

The UddiPing Example

The sample program `UddiPing.java` is another example of a standalone application. A Universal Description, Discovery and Integration (UDDI) service is a business registry and repository from which you can get information about busi-

nesses that have registered themselves with the registry service. For this example, the UddiPing application accesses a UDDI test registry to demonstrate a request being sent and a response being received. The application prints out the complete message that is returned, that is, the complete XML document as it looks when it comes over the wire. In addition to printing out the entire XML document, it also prints out just the text content of the response, making it much easier to see the information you want.

The files for the UddiPing example, in the directory `<INSTALL>/j2eetutorial/examples/jaxm/uddiping`, are `uddi.properties` and `UddiPing.java`.

You will be modifying the file `uddi.properties`, which contains the URL of the destination (the UDDI test registry) and the proxy host and proxy port of the sender. If you are not sure what the values for these are, you need to consult your system administrator or other person with that information.

Examining UddiPing

We will go through the file `UddiPing.java` a few lines at a time.

The first four lines of code import the packages used in the application.

```
import javax.xml.soap.*;
import javax.xml.messaging.*;
import java.util.*;
import java.io.*;
```

The next few lines begin the definition of the class `UddiPing`, which starts with the definition of its `main` method. The first thing it does is check to see if two arguments were supplied. If not, it prints a usage message and exits.

```
public class UddiPing {
    public static void main(String[] args) {
        try {
            if (args.length != 2) {
                System.err.println("Usage: MyUddiPing " +
                                   "properties-file business-name");
                System.exit(1);
            }
        }
    }
}
```

The following lines create a `java.util.Properties` file that contains the system properties and the properties from the file `uddi.properties` that is in the `uddiping` directory.

```
Properties myprops = new Properties();
myprops.load(new FileInputStream(args[0]));

Properties sysprops = System.getProperties();

Enumeration it = myprops.propertyNames();
while (it.hasMoreElements()) {
    String s = (String) it.nextElement();
    sysprops.setProperty(s,
        myprops.getProperty(s));
}
```

The next four lines create a `SOAPMessage` object. First, the code gets an instance of `SOAPConnectionFactory` and uses it to create a connection. Then it gets an instance of `MessageFactory` and uses it to create a message.

```
SOAPConnectionFactory scf =
    SOAPConnectionFactory.newInstance();
SOAPConnection connection =
    scf.createConnection();
MessageFactory msgFactory =
    MessageFactory.newInstance();
SOAPMessage msg = msgFactory.createMessage();
```

The new `SOAPMessage` object `msg` automatically contains a `SOAPPart` object that contains a `SOAPEnvelope` object. The `SOAPEnvelope` object contains a `SOAPBody` object, which is the element you want to access in order to add content to it. The next lines of code get the `SOAPPart` object, the `SOAPEnvelope` object, and the `SOAPBody` object.

```
SOAPEnvelope envelope =
    msg.getSOAPPart().getEnvelope();
SOAPBody body = envelope.getBody();
```

The following lines of code add an element with a fully-qualified name and then add two attributes to the new element. The first attribute has the name "generic" and the value "1.0". The second attribute has the name "maxRows" and the value "100". Then the code adds a child element with the name `name` and

adds some text to it with the method `addTextNode`. The text added is the business name you will supply when you run the application.

```
SOAPBodyElement findBusiness =
    body.addBodyElement(
        envelope.createName("find_business",
            "", "urn:uddi-org:api"));
findBusiness.addAttribute(
    envelope.createName("generic", "1.0");
findBusiness.addAttribute(
    envelope.createName("maxRows", "100");
SOAPElement businessName =
    findBusiness.addChildElement(
        envelope.createName("name"));
businessName.addTextNode(args[1]);
```

The next line of code creates the Java Object that represents the destination for this message. It gets the value of the property named "URL" from the system property file.

```
Object endpoint = new URLEndpoint(
    System.getProperties().getProperty("URL"));
```

The following line of code saves the changes that have been made to the message. This method will be called automatically when the message is sent, but it does not hurt to call it explicitly.

```
msg.saveChanges();
```

Next the message `msg` is sent to the destination that `endpoint` represents, which is the test UDDI registry. The method `call` will block until it gets a `SOAPMessage` object back, at which point it returns the reply.

```
SOAPMessage reply = connection.call(msg, endpoint);
```

In the next two lines, the first prints out a line giving the URL of the sender (the test registry), and the second prints out the returned message as an XML document.

```
System.out.println("Received reply from: " +
    endpoint);
reply.writeTo(System.out);
```

The remaining code makes the reply more user-friendly. It gets the content from certain elements rather than printing out the whole XML document as it was sent over the wire. Because the content is in the `SOAPBody` object, the first thing you need to do is access it, as shown in the following line of code. You can access each element in separate method calls, as was done in `Request.java`, or you can access the `SOAPBody` object using this shorthand version.

```
SOAPBody replyBody =  
    reply.getSOAPPart().getEnvelope().getBody();
```

Next you might print out two blank lines to separate your results from the raw XML message and a third line that describes the text that follows.

```
System.out.println("");  
System.out.println("");  
System.out.print(  
    "Content extracted from the reply message: ");
```

Now you can begin the process of getting all of the child elements from an element, getting the child elements from each of those, and so on, until you arrive at a text element that you can print out. Unfortunately, when you extract information from a registry, the number of subelements sometimes varies, making it difficult to know how many levels down the code needs to go. And in a test registry, there may be multiple entries for the same company name.

The code drills down through the subelements within the SOAP body and retrieves the name and description of the business. The method you use to retrieve child elements is the `SOAPElement` method `getChildElements`. When you give this method no arguments, it retrieves all of the child elements of the element on which it is called. If you know the `Name` object used to name an element, you can supply that to `getChildElements` and retrieve only the children with that name. In this example, however, you need to retrieve all elements and keep drilling down until you get to the elements that contain text content.

Here is the basic pattern that is repeated for drilling down:

```
Iterator iter1 = replyBody.getChildElements();  
while (iter1.hasNext()) {  
    SOAPBodyElement bodyElement =  
        (SOAPBodyElement)iter1.next();  
    Iterator iter2 =  
        bodyElement.getChildElements();  
    while (iter2.hasNext()) {  
        . . .  
    }  
}
```

The method `getChildElements` returns the elements in the form of a `java.util.Iterator` object. You access the child elements by calling the method `next` on the `Iterator` object. The method `Iterator.hasNext` can be used in a `while` loop because it returns `true` as long as the next call to the method `next` will return a child element. The loop ends when there are no more child elements to retrieve.

An immediate child of a `SOAPBody` object is a `SOAPBodyElement` object, which is why calling `iter1.next` returns a `SOAPBodyElement` object. Children of `SOAPBodyElement` objects and all child elements from there down are `SOAPElement` objects. For example, the call `iter2.next` returns the `SOAPElement` object `child2`. Note that the method `Iterator.next` returns an `Object`, which has to be narrowed (cast) to the specific kind of object you are retrieving. Thus, the result of calling `iter1.next` is cast to a `SOAPBodyElement` object, whereas the results of calling `iter2.next`, `iter3.next`, and so on, are all cast to a `SOAPElement` object.

Here is the code that prints out the business name and description and then ends the program:

```
Iterator iter1 = replyBody.getChildElements();
while (iter1.hasNext()) {
    SOAPBodyElement bodyElement =
        (SOAPBodyElement)iter1.next();
    Iterator iter2 =
        bodyElement.getChildElements();
    while (iter2.hasNext()) {
        SOAPElement child2 =
            (SOAPElement)iter2.next();
        Iterator iter3 =
            child2.getChildElements();
        String content = child2.getValue();
        System.out.println(content);
        while (iter3.hasNext()) {
            SOAPElement child3 =
                (SOAPElement)iter3.next();
            Iterator iter4 =
                child3.getChildElements();
            content = child3.getValue();
            System.out.println(content);
            while (iter4.hasNext()) {
                SOAPElement child4 =
                    (SOAPElement)iter4.next();
                content = child4.getValue();
                System.out.println(content);
            }
        }
    }
}
```



```

        }
    }
    }
    connection.close();
} catch (Exception ex) {
    ex.printStackTrace();
}
}
}

```

Editing the uddi.properties File

In order to run this example, you need to edit the `uddi.properties` file to specify a UDDI registry and the appropriate proxy host and port. Perform the following steps:

1. Mount the following filesystem by choosing Mount Filesystem from the File menu:

```
<INSTALL>/j2eetutorial/examples/jaxm/uddiping
```

INSTALL is the directory where you installed the Tutorial.

2. Right-click the `uddi.properties` file and choose the Edit menu item. The unedited file looks like this (the URLs are all on one line):

```
URL:http://uddi.ibm.com/testregistry/inquiryapi
#URL:http://localhost:8089/registry-server/
RegistryServerServlet
http.proxyHost:
http.proxyPort:8080
```

The default URL is the IBM test registry. If you wish, change the value of URL to another registry location. To use the Sun ONE Studio UDDI Server Registry, comment out the IBM registry and remove the comment character from the second line.

3. After `http.proxyHost`, insert the hostname and domain of your proxy server, if you access the Internet from behind a firewall. The format is usually *myhost.mydomain*.
4. If necessary, change the value `http.proxyPort` to one appropriate for your location. 8080 is the usual port number.
5. Save and close the file.

If you will be using the Sun ONE Studio UDDI Server Registry, start the registry if you have not already done so:

1. Select the Runtime tab of the Explorer.
2. Expand the UDDI Server Registry node.
3. Right-click the Internal UDDI Registry node and choose the Start Server menu item.

Compiling and Running UddiPing

To run the program, you need to specify two command-line arguments: the `uddi.properties` file and the name of the business for which you want to get a description.

Make sure you have followed the instructions in Setting the Classpath (page 324).

To compile and run UddiPing, perform the following steps:

1. Click the UddiPing file in the Filesystems tab.
2. In the property window for the file, choose the Execution tab.
3. Click Arguments, then double-click the ellipsis to bring up the property editor.
4. Enter the pathname of `uddi.properties`, a space, and `Oracle`. (If you are using the Sun ONE Studio UDDI Server Registry, enter a string from a business name that you know is in the registry.) On a Windows system, for example, you need to specify something like this (all on one line):

```
D:\j2eetutorial\examples\jaxm\uddiping\uddi.properties Oracle
```

5. Click OK.

If you are on a Windows system, notice that the pathname appears in the field with double backslashes.

6. Right-click the UddiPing file in the Filesystems tab and choose the Execute menu item. This command compiles the source file, then executes the class file.

The program output window displays the response message as follows (all on one line):

```
<?xml version="1.0" encoding="UTF-8" ?><Envelope
xmlns="http://schemas.xmlsoap.org/soap/envelope/"><Body>
<businessList generic="1.0" xmlns="urn:uddi-org:api"
operator="www.ibm.com/services/uddi"
truncated="false"><businessInfos><businessInfo
businessKey="26B46510-81E8-11D5-A4A5-0004AC49CC1E">
<name>Oracle</name><description xml:lang="en">oracle powers
the internet</description><serviceInfos></serviceInfos>
</businessInfo><businessInfo
businessKey="2ACAA2D0-82A7-11D5-A4A5-0004AC49CC1E">
<name>Oracle Corporation</name><description
xml:lang="en">Oracle Corporation provides the software and
services for
e-business.</description><serviceInfos><serviceInfo
serviceKey="4EDB6FC0-82AB-11D5-A4A5-0004AC49CC1E"
businessKey="2ACAA2D0-82A7-11D5-A4A5-0004AC49CC1E">
<name>E-Business Network</name></serviceInfo><serviceInfo
serviceKey="3AD5A9B0-82AA-11D5-A4A5-0004AC49CC1E"
businessKey="2ACAA2D0-82A7-11D5-A4A5-0004AC49CC1E">
<name>Oracle Store</name></serviceInfo><serviceInfo
serviceKey="0735C300-82AB-11D5-A4A5-0004AC49CC1E"
businessKey="2ACAA2D0-82A7-11D5-A4A5-0004AC49CC1E">
<name>Oracle Technology Network</name></serviceInfo>
<serviceInfo serviceKey="82757F80-82A9-11D5-A4A5-0004AC49CC1E"
businessKey="2ACAA2D0-82A7-11D5-A4A5-0004AC49CC1E">
<name>Oracle.com</name></serviceInfo></serviceInfos>
</businessInfo></businessInfos></businessList>
</Body></Envelope>
```

The following output appears after the full XML message.

Content extracted from the reply message:

Oracle
oracle powers the internet

Oracle Corporation
Oracle Corporation provides the software and services for e-
business.

There may be some occurrences of “null” in the output.

Running the program with Microsoft as the business-name property instead of Oracle produces the following output:

```
Received reply from:
http://uddi.ibm.com/testregistry/inquiryapi
<?xml version="1.0" encoding="UTF-8" ?><Envelope
xmlns="http://schemas.xmlsoap.org/soap/envelope/"><Body>
<businessList generic="1.0" xmlns="urn:uddi-org:api"
operator="www.ibm.com/services/uddi"
truncated="false"><businessInfos><businessInfo
businessKey="D7475060-BF58-11D5-A432-0004AC49CC1E">
<name>Microsoft Corporation</name><description
xml:lang="en">Computer Software and Hardware
Manufacturer</description><serviceInfos></serviceInfos>
</businessInfo></businessInfos></businessList>
</Body></Envelope>
```

Content extracted from the reply message:

```
Microsoft Corporation
Computer Software and Hardware Manufacturer
```

The SOAPFaultTest Example

The code `SOAPFaultTest.java`, based on the code fragments in a preceding section (SOAP Faults, page 319) creates a message with a `SOAPFault` object. It then retrieves the contents of the `SOAPFault` object and prints them out. You will find the code for `SOAPFaultTest` in the following directory:

```
<INSTALL>/j2eetutorial/examples/jaxm/fault
```

Here is the file `SOAPFaultTest.java`.

```
import javax.xml.soap.*;
import java.util.*;

public class SOAPFaultTest {

    public static void main(String[] args) {
        try {
            MessageFactory msgFactory =
                MessageFactory.newInstance();
            SOAPMessage msg = msgFactory.createMessage();
            SOAPEnvelope envelope =
                msg.getSOAPPart().getEnvelope();
```

```

SOAPBody body = envelope.getBody();
SOAPFault fault = body.addFault();

fault.setFaultCode("Client");
fault.setFaultString(
    "Message does not have necessary info");
fault.setFaultActor("http://gizmos.com/order");

Detail detail = fault.addDetail();

Name entryName = envelope.createName("order",
    "PO", "http://gizmos.com/orders/");
DetailEntry entry =
    detail.addDetailEntry(entryName);
entry.addTextNode(
    "quantity element does not have a value");

Name entryName2 =
    envelope.createName("confirmation", "PO",
        "http://gizmos.com/confirm");
DetailEntry entry2 =
    detail.addDetailEntry(entryName2);
entry2.addTextNode("Incomplete address: " +
    "no zip code");

msg.saveChanges();

// Now retrieve the SOAPFault object and
// its contents, after checking to see that
// there is one
System.out.println(
    "Here is what the XML message looks like:");
msg.writeTo(System.out);
System.out.println();
System.out.println();

if ( body.hasFault() ) {
    fault = body.getFault();
    String code = fault.getFaultCode();
    String string = fault.getFaultString();
    String actor = fault.getFaultActor();

    System.out.println("SOAP fault contains: ");
    System.out.println("  fault code = " + code);
    System.out.println("  fault string = " +
        string);
    if ( actor != null) {
        System.out.println("  fault actor = " +

```

```

        actor);
    }

    detail = fault.getDetail();
    if ( detail != null) {
        Iterator it = detail.getDetailEntries();
        while ( it.hasNext() ) {
            entry = (DetailEntry)it.next();
            String value = entry.getValue();
            System.out.println(
                "    Detail entry = " + value);
        }
    }
}
} catch (Exception ex) {
    ex.printStackTrace();
}
}
}

```

Compiling and Running SOAPFaultTest.java

To compile and run `SoapFaultTest`, first make sure you followed the instructions in *Setting the Classpath* (page 324). Then perform the following steps:

1. Mount the following filesystem:

```
<INSTALL>/j2eetutorial/examples/jaxm/fault
```

2. Right-click the `SoapFaultTest` file in the Filesystems tab and choose the Execute menu item.

The program output window displays the response message as follows (the SOAP message is all on one line):

Here is what the XML message looks like:

```

<?xml version="1.0" encoding="UTF-8"?>
<soap-env:Envelope xmlns:soap-
env="http://schemas.xmlsoap.org/soap/envelope/">
<soap-env:Header/><soap-env:Body>
<soap-env:Fault><faultcode>Client</faultcode>
<faultstring>Message does not have necessary info</faultstring>
<faultactor>http://gizmos.com/order</faultactor>
<detail><PO:order
xmlns:PO="http://gizmos.com/orders/">quantity element does not
have a value</PO:order><PO:confirmation
xmlns:PO="http://gizmos.com/confirm">Incomplete address: no

```

```
zip code</PO:confirmation></detail></soap-env:Fault>
</soap-env:Body></soap-env:Envelope>
```

SOAP fault contains:

```
fault code = Client
fault string = Message does not have necessary info
fault actor = http://gizmos.com/order
Detail entry = quantity element does not have a value
Detail entry = Incomplete address: no zip code
```

Starting the Application Server

The UddiPing and SOAPFaultTest examples are standalone programs; that is, they do not run in a container. The remaining examples run in a container. If you have not already done so, start the Sun ONE Application Server now and specify it as the default server for Web Tier Applications.

The JAXM Simple Example

The JAXM Simple example shows how to send and receive a message using the local provider. Note that a *local provider* should not be confused with a messaging provider. The local provider is simply a mechanism for returning the reply to a message that was sent using the method `SOAPConnection.call`. Note that a message sent by this method will always be a request-response message. Running this example generates the files `sent.msg` and `reply.msg`, which you will find in the server instance configuration directory.

To deploy and run the JAXM Simple example, perform the following steps:

1. Mount the following filesystem:

```
<INSTALL>/j2eetutorial/examples/jaxm/jaxm-simple
```

2. In the Mount Web Module dialog that appears, click OK to display the alternate view of the web module.
3. An information dialog may appear stating that `ReceivingServlet.java` cannot be converted to a servlet. Click OK.
4. Specify the following context root for the web module in the Context Root field of the WEB-INF property window:

```
/jaxm-simple
```

5. Right-click the WEB-INF node and choose Deploy from the menu.
6. Right-click the WEB-INF node and choose Execute from the menu. If this doesn't work, open a web browser and enter the following URL:

`localhost:80/jaxm-simple/`

If the page cannot be found, make sure the server is running.

A web page appears with the following text:

This is a simple example of a round-trip JAXM message exchange.

Click [here](#) to send the message.

If you click the link, another page appears with the following text:

Sent message (check “sent.msg”) and received reply (check “reply.msg”).

You can find the files `sent.msg` and `reply.msg` in the following directory:

`<S1AS7_HOME>/domains/domain1/server1/config`

The SAAJ Simple Example

The SAAJ Simple application is similar to the Simple example except that it is written using only the SAAJ API. In SAAJ Simple, the `call` method takes a Java Object rather than a `URLEndpoint` object to designate the recipient, and thus uses only the `javax.xml.soap` package. Running this example generates the files `sent.msg` and `reply.msg`, which you will find in the server instance configuration directory.

To deploy and run the SAAJ Simple example, perform the following steps:

1. Mount the following filesystem:

`<INSTALL>/j2eetutorial/examples/jaxm/saaj-simple`

2. An information dialog may appear stating that `ReceivingServlet.java` cannot be converted to a servlet. Click OK.
3. Specify the following context root for the web module:

`/saaj-simple`

4. Right-click the WEB-INF node and choose Deploy from the menu.

5. Right-click the WEB-INF node and choose Execute from the menu. If this doesn't work, open a web browser and enter the following URL:

```
localhost:80/saa-j-simple/index.html
```

A web page appears with the following text:

This is a simple example of a roundtrip SAAJ message exchange.

Click [here](#) to send the message.

If you click the link, another page appears with the following text:

Sent message (check "sent.msg") and received reply (check "reply.msg").

You can find the files `sent.msg` and `reply.msg` in the following directory:

```
<S1AS7_HOME>/domains/domain1/server1/config
```

The JAXM Translator Example

The JAXM Translator example uses a simple translation service to translate a given word into different languages. If you have given the correct proxy host and proxy port, the word you supply will be translated into French, German, and Italian. Running this example generates the files `request.msg` and `reply.msg` in the server configuration directory.

Make sure you have followed the instructions in Changing Server Permissions (page 325) before you begin.

To deploy and run the JAXM Translator example, perform the following steps:

1. Mount the following filesystem:

```
<INSTALL>/j2eetutorial/examples/jaxm/jaxm-translator
```

2. An information dialog may appear stating that `ReceivingServlet.java` cannot be converted to a servlet. Click OK.
3. Specify the following context root for the web module:

```
/jaxm-translator
```

4. Right-click the WEB-INF node and choose Deploy from the menu.
5. Right-click the WEB-INF node and choose Execute from the menu. If this doesn't work, open a web browser and enter the following URL:

`localhost:80/jaxm-translator/index.html`

A web page with the header “Translator Sample Application” appears.

6. Enter the proxy host and port for your system in the text fields provided.
7. Enter the text to be translated in the field provided.
8. Choose the In SOAPBody or As Attachments radio button and click Translate.
9. A web page appears with the word translated into the three languages.
10. Use the browser’s back button to run the example again.

Check `reply.msg` after getting the reply in the SOAP body and again after getting the reply as an attachment to see the difference in what is sent as a reply.

You can find the files `request.msg` and `reply.msg` in the following directory:

`<S1AS7_HOME>/domains/domain1/server1/config`

The JAXM Tags Example

The JAXM Tags example uses JavaServer Pages tags to generate and consume a SOAP message.

Make sure you have followed the instructions in Changing Server Permissions (page 325) before you begin.

To deploy and run the JAXM Tags example, perform the following steps:

1. Mount the following filesystem:

`<INSTALL>/j2eetutorial/examples/jaxm/jaxm-tags`

2. Specify the following context root for the web module:

`/jaxm-tags`

3. Right-click the WEB-INF node and choose Deploy from the menu.
4. Right-click the WEB-INF node and choose Execute from the menu. If this doesn’t work, open a web browser and enter the following URL:

`localhost:80/jaxm-tags/index.html`

If the page cannot be found, make sure the server is running.

5. A web page with the header “JSP Examples” appears. Click on each of the three links. For each, a page appears with the requests and responses.

The JAXM Provider

The JAXM Provider is a simple example of a JAXM provider.

The source files for the JAXM Provider, like those for the JAXM Provider Administrator, are in the directory `<S1STUDIO_HOME>/jwsdp/services/`.

Before you deploy the provider, use the application server administration tool to add `<S1STUDIO_HOME>/jwsdp/common/lib/jaxm-runtime.jar` to the application server classpath suffix and restart the application server. To do so:

1. Open the URL `http://localhost:4848` in a browser
2. Select the `server1` node.
3. Select the JVM Settings tab.
4. Click the Path Settings link.
5. Add `<S1STUDIO_HOME>/jwsdp/common/lib/jaxm-runtime.jar` to the Classpath Suffix text area.
6. Click Save.
7. Click the General tab.
8. Apply the changes, then stop and restart the server.

To deploy and run the JAXM Provider, perform the following steps:

1. Mount the following filesystem:
`<S1STUDIO_HOME>/jwsdp/services/jaxm-provider`

If you click on the file `WEB-INF/provider.xml`, you may notice that it is marked as having invalid XML. If you open this file in a text editor, right-click, and choose Validate XML from the menu, you will see an error message about a missing DTD file. Ignore this error message.

2. Specify the following context root for the web module:

`/jaxm-provider`

3. Right-click the `WEB-INF` node and choose Deploy from the menu.

The JAXM Provider Administrator

The JAXM Provider Administrator is a simple web-based administrative tool for the JAXM provider.

The source files for the JAXM Provider Administrator are not in the tutorial examples directory. Instead, they are in `<S1STUDIO_HOME>/jwsdp/services/`.

To deploy and run the JAXM Provider Administrator, perform the following steps:

1. Mount the following filesystem:

```
<S1STUDIO_HOME>/jwsdp/services/jaxm-provideradmin
```

If you click on the file `WEB-INF/provider.xml`, you may notice that it is marked as having invalid XML. If you open this file in a text editor, right-click, and choose `Validate XML` from the menu, you will see an error message about a missing DTD file. Ignore this error message.

2. In order to run the JAXM Provider Administrator example using the Sun ONE Application Server, you need to change the security properties for the `web.xml` file as follows:
 - a. Expand the `WEB-INF` node if you have not already done so.
 - b. Click the `web.xml` file.
 - c. In the property window for the file, click the `Security` tab. You will see that the `Security Constraints` field contains the text “1 Security Constraint”. Click this field, then click the ellipsis.
 - d. In the `Property Editor`, select the security constraint and click `Remove`.
 - e. Click `OK`.

3. Specify the following context root for the web module:

```
/jaxm-provideradmin
```

4. Right-click the `WEB-INF` node and choose `Deploy` from the menu.
5. Right-click the `WEB-INF` node and choose `Execute` from the menu. If this doesn't work, open a web browser and enter the following URL:

```
localhost:80/jaxm-provideradmin/
```

The JAXM Provider Administration Tool appears. Expand the nodes in the left window to see what you can do in the tool. See the next section for more information.

Using the JAXM Provider Administration Tool

The Provider Administration tool is a convenient means of configuring a messaging provider. A messaging provider, a third party service, handles the behind-the-scenes details of the routing and transmission of JAXM messages. For more information about messaging providers, see *Messaging Providers* (page 298).

The Provider Administration tool is normally used by System Administrators, but others may use it as well. Exploring this tool gives you more of an idea of what a messaging provider needs to know. For example, a messaging provider maintains a list of the endpoints to which you can send messages. You can add a new endpoint to this list using the Provider Administration tool. If a message is not delivered successfully on the first try, a messaging provider will continue attempting to deliver it. You can specify the number of times the messaging provider should attempt delivery by supplying a retry limit. Setting this limit is another thing you can do with the Provider Administration tool.

The following lists the ways you can use the tool to set a messaging provider's properties.

- To add, modify, or delete an endpoint
- To change the number of retries (the number of times the provider will try to send a message)
- To change the retry interval (the amount of time the provider will wait before trying to send a message again)
- To change the directory where the provider logs messages
- To set the number of messages per log file

Conclusion

JAXM provides a Java API that simplifies writing and sending XML messages. You have seen how to use this API to write client code for JAXM request-response messages and one-way messages. You have also seen how to get the content from a reply message. This knowledge was applied in writing and running the `UddiPing` and `SOAPFaultTest` examples. You have also learned how to deploy simple JAXM examples in a J2EE server. In addition, the case study (The Coffee Break Application, page 383) provides detailed examples of JAXM code for both the client and server.

You now have first-hand experience of how JAXM makes it easier to do XML messaging.

Further Information

You can find additional information about JAXM from the following:

- Documents bundled with the JAXM Reference Implementation at

`<S1STUDIO_HOME>/jwsdp/docs/jaxm/`

- SAAJ 1.1 specification, available from

`http://java.sun.com/xml/downloads/saaaj.html`

- JAXM 1.1 specification, available from

`http://java.sun.com/xml/downloads/jaxm.html`

- JAXM website at

`http://java.sun.com/xml/jaxm/`

Publishing and Discovering Web Services with JAXR

Kim Haase

THE Java API for XML Registries (JAXR) provides a uniform and standard Java API for accessing different kinds of XML registries.

After providing a brief overview of JAXR, this chapter describes how to implement a JAXR client to publish an organization and its web services to a registry and to query a registry to find organizations and services. Finally, it explains how to run the examples provided with this tutorial and offers links to more information on JAXR.

Overview of JAXR

This section covers the following topics:

- What is a registry?
- What is JAXR?
- JAXR architecture

What Is a Registry?

An XML *registry* is an infrastructure that enables the building, deployment, and discovery of Web services. It is a neutral third party that facilitates dynamic and loosely coupled business-to-business (B2B) interactions. A registry is available to organizations as a shared resource, often in the form of a Web-based service.

Currently there are a variety of specifications for XML registries. These include

- The ebXML Registry and Repository standard, which is sponsored by the Organization for the Advancement of Structured Information Standards (OASIS) and the United Nations Centre for the Facilitation of Procedures and Practices in Administration, Commerce and Transport (U.N./CEFACT)
- The Universal Description, Discovery, and Integration (UDDI) project, which is being developed by a vendor consortium

A *registry provider* is an implementation of a business registry that conforms to a specification for XML registries.

What Is JAXR?

JAXR enables Java software programmers to use a single, easy-to-use abstraction API to access a variety of XML registries. A unified JAXR information model describes content and metadata within XML registries.

JAXR gives developers the ability to write registry client programs that are portable across different target registries. JAXR also enables value-added capabilities beyond those of the underlying registries.

The current version of the JAXR specification includes detailed bindings between the JAXR information model and both the ebXML Registry and the UDDI version 2 specifications. You can find the latest version of the specification at

<http://java.sun.com/xml/downloads/jaxr.html>

At this release, JAXR implements the level 0 capability profile defined by the JAXR specification. This level allows access to both UDDI and ebXML registries at a basic level. At this release, JAXR supports access only to UDDI version 2 registries.

Currently several public UDDI version 2 registries exist.

The Sun ONE Studio internal UDDI Server Registry provides a UDDI version 2 registry that you can use to test your JAXR applications in a private environment. The Registry does not support messages defined in the UDDI Version 2.0 Replication Specification.

Several ebXML registries are under development, and one is available at the Center for E-Commerce Infrastructure Development (CECID), Department of Computer Science Information Systems, The University of Hong Kong (HKU). For information, see <http://www.cec.id.hku.hk/Release/PR09APR2002.html>.

A JAXR provider for ebXML registries is available in open source at <http://ebxmlrr.sourceforge.net>.

JAXR Architecture

The high-level architecture of JAXR consists of the following parts:

- A *JAXR client*: a client program that uses the JAXR API to access a business registry via a JAXR provider.
- A *JAXR provider*: an implementation of the JAXR API that provides access to a specific registry provider or to a class of registry providers that are based on a common specification.

A JAXR provider implements two main packages:

- `javax.xml.registry`, which consists of the API interfaces and classes that define the registry access interface.
- `javax.xml.registry.infomodel`, which consists of interfaces that define the information model for JAXR. These interfaces define the types of objects that reside in a registry and how they relate to each other. The basic interface in this package is the `RegistryObject` interface. Its subinterfaces include `Organization`, `Service`, and `ServiceBinding`.

The most basic interfaces in the `javax.xml.registry` package are

- `Connection`. The `Connection` interface represents a client session with a registry provider. The client must create a connection with the JAXR provider in order to use a registry.
- `RegistryService`. The client obtains a `RegistryService` object from its connection. The `RegistryService` object in turn enables the client to obtain the interfaces it uses to access the registry.

The primary interfaces, also part of the `javax.xml.registry` package, are

- `BusinessQueryManager`, which allows the client to search a registry for information in accordance with the `javax.xml.registry.infomodel` interfaces. An optional interface, `DeclarativeQueryManager`, allows the client to use SQL syntax for queries. (The implementation of JAXR in the Sun ONE Application Server does not implement `DeclarativeQueryManager`.)
- `BusinessLifeCycleManager`, which allows the client to modify the information in a registry by either saving it (updating it) or deleting it.

When an error occurs, JAXR API methods throw a `JAXRException` or one of its subclasses.

Many methods in the JAXR API use a `Collection` object as an argument or a returned value. Using a `Collection` object allows operations on several registry objects at a time.

Figure 11–1 illustrates the architecture of JAXR. In the Sun ONE Application Server, a JAXR client uses the capability level 0 interfaces of the JAXR API to access the JAXR provider. The JAXR provider in turn accesses a registry. The Sun ONE Application Server supplies a JAXR provider for UDDI registries.

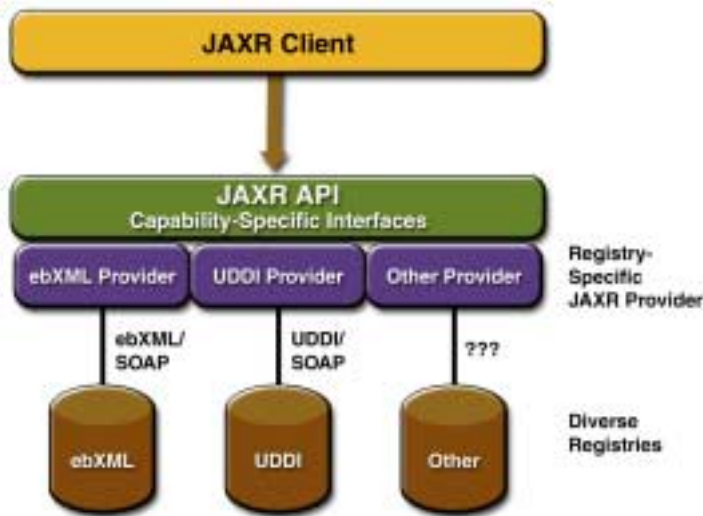


Figure 11–1 JAXR Architecture

Implementing a JAXR Client

This section describes the basic steps to follow in order to implement a JAXR client that can perform queries and updates to a UDDI registry. A JAXR client is a client program that can access registries using the JAXR API.

This tutorial does not describe how to implement a JAXR provider. A JAXR provider provides an implementation of the JAXR specification that allows access to an existing registry provider, such as a UDDI or ebXML registry. The implementation of JAXR in the Sun ONE Application Server itself is an example of a JAXR provider.

This tutorial includes several client examples, which are described in Running the Client Examples (page 371). The examples are in the directory `<INSTALL>/j2eetutorial/examples/jaxr`, where `<INSTALL>` is the directory where you installed the tutorial.

Establishing a Connection

The first task a JAXR client must complete is to establish a connection to a registry.

Preliminaries: Getting Access to a Registry

Any user of a JAXR client may perform queries on a registry. In order to add data to the registry or to update registry data, however, a user must obtain permission from the registry to access it. To register with one of the public UDDI version 2 registries, go to one of the following Web sites and follow the instructions:

- <http://test.uddi.microsoft.com/> (Microsoft)
- <http://uddi.ibm.com/testregistry/registry.html> (IBM)
- <http://udditest.sap.com/> (SAP)

These UDDI version 2 registries are intended for testing purposes. When you register, you will obtain a user name and password. You will specify this user name and password for some of the JAXR client example programs.

You do not have to register with the Sun ONE Studio internal UDDI Server Registry in order to add or update data. You can use the default user name and password, `testuser` and `testuser`.

Note: The JAXR API has been tested with the Microsoft and IBM registries and with the Registry Server, but not with the SAP registry.

Creating or Looking Up a Connection Factory

A client creates a connection from a connection factory. A JAXR provider may supply one or more preconfigured connection factories that clients can obtain by looking them up using the Java Naming and Directory Interface™ (JNDI) API.

At this release of the Sun ONE Application Server, JAXR does not supply pre-configured connection factories. Instead, a client creates an instance of the abstract class `ConnectionFactory`:

```
import javax.xml.registry.*;
...
ConnectionFactory connFactory =
    ConnectionFactory.newInstance();
```

Creating a Connection

To create a connection, a client first creates a set of properties that specify the URL or URLs of the registry or registries being accessed. For example, the following code provides the URLs of the query service and publishing service for the IBM test registry. (There should be no line break in the strings.)

```
Properties props = new Properties();
props.setProperty("javax.xml.registry.queryManagerURL",
    "http://uddi.ibm.com/testregistry/inquiryapi");
props.setProperty("javax.xml.registry.lifeCycleManagerURL",
    "https://uddi.ibm.com/testregistry/protect/publishapi");
```

With the Sun ONE Application Server implementation of JAXR, if the client is accessing a registry that is outside a firewall, it must also specify proxy host and port information for the network on which it is running. For queries it may need to specify only the HTTP proxy host and port; for updates it must specify the HTTPS proxy host and port.

```
props.setProperty("com.sun.xml.registry.http.proxyHost",
    "myhost.mydomain");
props.setProperty("com.sun.xml.registry.http.proxyPort",
    "8080");
```

```
props.setProperty("com.sun.xml.registry.https.proxyHost",
    "myhost.mydomain");
props.setProperty("com.sun.xml.registry.https.proxyPort",
    "8080");
```

The client then sets the properties for the connection factory and creates the connection:

```
connFactory.setProperties(props);
Connection connection = connFactory.createConnection();
```

The `makeConnection` method in the sample programs shows the steps used to create a JAXR connection.

Setting Connection Properties

The implementation of JAXR in the Sun ONE Application Server allows you to set a number of properties on a JAXR connection. Some of these are standard properties defined in the JAXR specification. Other properties are specific to the implementation of JAXR in the Sun ONE Application Server. Table 11–1 and Table 11–2 list and describe these properties.

Table 11–1 Standard JAXR Connection Properties

Property Name and Description	Data Type	Default Value
javax.xml.registry.queryManagerURL Specifies the URL of the query manager service within the target registry provider	String	None
javax.xml.registry.lifeCycleManagerURL Specifies the URL of the life cycle manager service within the target registry provider (for registry updates)	String	Same as the specified queryManagerURL value
javax.xml.registry.semanticEquivalences Specifies semantic equivalences of concepts as one or more tuples of the ID values of two equivalent concepts separated by a comma; the tuples are separated by vertical bars: id1,id2 id3,id4	String	None

Table 11–1 Standard JAXR Connection Properties

Property Name and Description	Data Type	Default Value
javax.xml.registry.security.authentication-Method Provides a hint to the JAXR provider on the authentication method to be used for authenticating with the registry provider	String	None; UDDI_GET_AUTH_TOKEN is the only supported value
javax.xml.registry.uddi.maxRows The maximum number of rows to be returned by find operations. Specific to UDDI providers	Integer	None
javax.xml.registry.postalAddressScheme The ID of a ClassificationScheme to be used as the default postal address scheme. See Specifying Postal Addresses (page 369) for an example	String	None

Table 11–2 Implementation-Specific JAXR Connection Properties

Property Name and Description	Data Type	Default Value
com.sun.xml.registry.http.proxyHost Specifies the HTTP proxy host to be used for accessing external registries	String	None
com.sun.xml.registry.http.proxyPort Specifies the HTTP proxy port to be used for accessing external registries; usually 8080	String	None
com.sun.xml.registry.https.proxyHost Specifies the HTTPS proxy host to be used for accessing external registries	String	Same as HTTP proxy host value
com.sun.xml.registry.https.proxyPort Specifies the HTTPS proxy port to be used for accessing external registries; usually 8080	String	Same as HTTP proxy port value

Table 11–2 Implementation-Specific JAXR Connection Properties

Property Name and Description	Data Type	Default Value
com.sun.xml.registry.http.proxyUserName Specifies the user name for the proxy host for HTTP proxy authentication, if one is required	String	None
com.sun.xml.registry.http.proxyPassword Specifies the password for the proxy host for HTTP proxy authentication, if one is required	String	None
com.sun.xml.registry.useCache Tells the JAXR implementation to look for registry objects in the cache first and then to look in the registry if not found	Boolean, passed in as String	True
com.sun.xml.registry.useSOAP Tells the JAXR implementation to use Apache SOAP rather than the Java API for XML Messaging; may be useful for debugging. You must install Apache SOAP (downloadable from the Apache web site, http://xml.apache.org/) in order to use this option	Boolean, passed in as String	False

You can set these properties as follows:

- Most of these properties must be set as connection properties in a JAXR client program. For example:

```
Properties props = new Properties();
props.setProperty("javax.xml.registry.queryManagerURL",
    "http://uddi.ibm.com/testregistry/inquiryapi");
props.setProperty("javax.xml.registry.lifeCycleManagerURL",
    "https://uddi.ibm.com/testregistry/protect/publishapi");
ConnectionFactory factory = ConnectionFactory.newInstance();
factory.setProperties(props);
connection = factory.createConnection();
```

- The postalAddressScheme, useCache, and useSOAP properties may be set as system properties in the client program.

An additional system property specific to the implementation of JAXR in the Sun ONE Application Server is `com.sun.xml.registry.userTaxonomyFileNames`. For details on using this property, see *Defining a Taxonomy* (page 366).

Obtaining and Using a RegistryService Object

After creating the connection, the client uses the connection to obtain a `RegistryService` object and then the interface or interfaces it will use:

```
RegistryService rs = connection.getRegistryService();
BusinessQueryManager bqm = rs.getBusinessQueryManager();
BusinessLifeCycleManager blcm =
    rs.getBusinessLifeCycleManager();
```

Typically, a client obtains both a `BusinessQueryManager` object and a `BusinessLifeCycleManager` object from the `RegistryService` object. If it is using the registry for simple queries only, it may need to obtain only a `BusinessQueryManager` object.

Querying a Registry

The simplest way for a client to use a registry is to query it for information about the organizations that have submitted data to it. The `BusinessQueryManager` interface supports a number of find methods that allow clients to search for data using the JAXR information model. Many of these methods return a `BulkResponse` (a collection of objects) that meets a set of criteria specified in the method arguments. The most useful of these methods are:

- `findOrganizations`, which returns a list of organizations that meet the specified criteria—often a name pattern or a classification within a classification scheme
- `findServices`, which returns a set of services offered by a specified organization
- `findServiceBindings`, which returns the service bindings (information about how to access the service) that are supported by a specified service

The `JAXRQuery` program illustrates how to query a registry by organization name and display the data returned. The `JAXRQueryByNAICSClassification` and `JAXRQueryByWSDLClassification` programs illustrate how to query a registry

using classifications. All JAXR providers support at least the following taxonomies for classifications:

- The North American Industry Classification System (NAICS). See <http://www.census.gov/epcd/www/naics.html> for details.
- The Universal Standard Products and Services Classification (UNSPSC). See <http://www.eccma.org/unspsc/> for details.
- The ISO 3166 country codes classification system maintained by the International Organization for Standardization (ISO). See <http://www.iso.org/iso/en/prods-services/iso3166ma/index.html> for details.

The following sections describe how to perform some common queries.

Finding Organizations by Name

To search for organizations by name, you normally use a combination of find qualifiers (which affect sorting and pattern matching) and name patterns (which specify the strings to be searched). The `findOrganizations` method takes a collection of `findQualifier` objects as its first argument and a collection of `namePattern` objects as its second argument. The following fragment shows how to find all the organizations in the registry whose names begin with a specified string, `qString`, and to sort them in alphabetical order.

```
// Define find qualifiers and name patterns
Collection findQualifiers = new ArrayList();
findQualifiers.add(FindQualifier.SORT_BY_NAME_DESC);
Collection namePatterns = new ArrayList();
namePatterns.add(qString);

// Find using the name
BulkResponse response =
    bqm.findOrganizations(findQualifiers,
        namePatterns, null, null, null, null);
Collection orgs = response.getCollection();
```

A client can use percent signs (%) to specify that the query string can occur anywhere within the organization name. For example, the following code fragment

performs a case-sensitive search for organizations whose names contain `qString`:

```
Collection findQualifiers = new ArrayList();
findQualifiers.add(FindQualifier.CASE_SENSITIVE_MATCH);
Collection namePatterns = new ArrayList();
namePatterns.add("%" + qString + "%");

// Find orgs with name containing qString
BulkResponse response =
    bqm.findOrganizations(findQualifiers, namePatterns, null,
        null, null, null);
Collection orgs = response.getCollection();
```

Finding Organizations by Classification

To find organizations by classification, you need to establish the classification within a particular classification scheme and then specify the classification as an argument to the `findOrganizations` method.

The following code fragment finds all organizations that correspond to a particular classification within the NAICS taxonomy. (You can find the NAICS codes at <http://www.census.gov/epcd/naics/naicscod.txt>.)

```
ClassificationScheme cScheme =
    bqm.findClassificationSchemeByName(null,
        "ntis-gov:naics");
Classification classification =
    blcm.createClassification(cScheme,
        "Snack and Nonalcoholic Beverage Bars", "722213");
Collection classifications = new ArrayList();
classifications.add(classification);
// make JAXR request
BulkResponse response = bqm.findOrganizations(null,
    null, classifications, null, null, null);
Collection orgs = response.getCollection();
```

You can also use classifications to find organizations that offer services based on technical specifications that take the form of WSDL (Web Services Description Language) documents. In JAXR, a concept is used as a proxy to hold the information about a specification. The steps are a little more complicated than in the previous example, because the client must find the specification concepts first, then the organizations that use those concepts.

The following code fragment finds all the WSDL specification instances used within a given registry. You can see that the code is similar to the NAICS query code except that it ends with a call to `findConcepts` instead of `findOrganizations`.

```
String schemeName = "uddi-org:types";
ClassificationScheme uddiOrgTypes =
    bqmc.findClassificationSchemeByName(null, schemeName);

/*
 * Create a classification, specifying the scheme
 * and the taxonomy name and value defined for WSDL
 * documents by the UDDI specification.
 */
Classification wsdlSpecClassification =
    blcm.createClassification(uddiOrgTypes,
        "wsdlSpec", "wsdlSpec");

Collection classifications = new ArrayList();
classifications.add(wsdlSpecClassification);

// Find concepts
BulkResponse br = bqmc.findConcepts(null, null,
    classifications, null, null);
```

To narrow the search, you could use other arguments of the `findConcepts` method (search qualifiers, names, external identifiers, or external links).

The next step is to go through the concepts, find the WSDL documents they correspond to, and display the organizations that use each document:

```
// Display information about the concepts found
Collection specConcepts = br.getCollection();
Iterator iter = specConcepts.iterator();
if (!iter.hasNext()) {
    System.out.println("No WSDL specification concepts found");
} else {
    while (iter.hasNext()) {
        Concept concept = (Concept) iter.next();

        String name = getName(concept);

        Collection links = concept.getExternalLinks();
        System.out.println("\nSpecification Concept:\n\tName: " +
            name + "\n\tKey: " +
            concept.getKey().getId() +
            "\n\tDescription: " +
```

```

        getDescription(concept));
    if (links.size() > 0) {
        ExternalLink link =
            (ExternalLink) links.iterator().next();
        System.out.println("\tURL of WSDL document: '" +
            link.getExternalURI() + "'");
    }

    // Find organizations that use this concept
    Collection specConcepts1 = new ArrayList();
    specConcepts1.add(concept);
    br = bqm.findOrganizations(null, null, null,
        specConcepts1, null, null);

    // Display information about organizations
    ...
}

```

If you find an organization that offers a service you wish to use, you can invoke the service using the JAX-RPC API.

Finding Services and ServiceBindings

After a client has located an organization, it can find that organization's services and the service bindings associated with those services.

```

    Iterator orgIter = orgs.iterator();
    while (orgIter.hasNext()) {
        Organization org = (Organization) orgIter.next();
        Collection services = org.getServices();
        Iterator svcIter = services.iterator();
        while (svcIter.hasNext()) {
            Service svc = (Service) svcIter.next();
            Collection serviceBindings =
                svc.getServiceBindings();
            Iterator sbIter = serviceBindings.iterator();
            while (sbIter.hasNext()) {
                ServiceBinding sb =
                    (ServiceBinding) sbIter.next();
            }
        }
    }
}

```

Managing Registry Data

If a client has authorization to do so, it can submit data to a registry, modify it, and remove it. It uses the `BusinessLifeCycleManager` interface to perform these tasks.

Registries usually allow a client to modify or remove data only if the data is being modified or removed by the same user who first submitted the data.

Getting Authorization from the Registry

Before it can submit data, the client must send its user name and password to the registry in a set of credentials. The following code fragment shows how to do this.

```
String username = "myUserName";
String password = "myPassword";

// Get authorization from the registry
PasswordAuthentication passwdAuth =
    new PasswordAuthentication(username,
        password.toCharArray());

Set creds = new HashSet();
creds.add(passwdAuth);
connection.setCredentials(creds);
```

Creating an Organization

The client creates the organization and populates it with data before saving it.

An `Organization` object is one of the more complex data items in the JAXR API. It normally includes the following:

- A `Name` object
- A `Description` object
- A `Key` object, representing the ID by which the organization is known to the registry. This key is created by the registry, not by the user, and is returned after the organization is submitted to the registry.
- A `PrimaryContact` object, which is a `User` object that refers to an authorized user of the registry. A `User` object normally includes a `PersonName`

object and collections of `TelephoneNumber`, `EmailAddress`, and/or `PostalAddress` objects.

- A collection of `Classification` objects
- Service objects and their associated `ServiceBinding` objects

For example, the following code fragment creates an organization and specifies its name, description, and primary contact. When a client creates an organization, it does not include a key; the registry returns the new key when it accepts the newly created organization. The `blcm` object in this code fragment is the `BusinessLifeCycleManager` object returned in *Obtaining and Using a Registry-Service Object* (page 356). An `InternationalString` object is used for string values that may need to be localized.

```
// Create organization name and description
Organization org =
    blcm.createOrganization("The Coffee Break");
InternationalString s =
    blcm.createInternationalString("Purveyor of " +
        "the finest coffees. Established 1914");
org.setDescription(s);

// Create primary contact, set name
User primaryContact = blcm.createUser();
PersonName pName = blcm.createPersonName("Jane Doe");
primaryContact.setPersonName(pName);

// Set primary contact phone number
TelephoneNumber tNum = blcm.createTelephoneNumber();
tNum.setNumber("(800) 555-1212");
Collection phoneNums = new ArrayList();
phoneNums.add(tNum);
primaryContact.setTelephoneNumbers(phoneNums);

// Set primary contact email address
EmailAddress emailAddress =
    blcm.createEmailAddress("jane.doe@TheCoffeeBreak.com");
Collection emailAddresses = new ArrayList();
emailAddresses.add(emailAddress);
primaryContact.setEmailAddresses(emailAddresses);

// Set primary contact for organization
org.setPrimaryContact(primaryContact);
```

Adding Classifications

Organizations commonly belong to one or more classifications based on one or more classification schemes (taxonomies). To establish a classification for an organization using a taxonomy, the client first locates the taxonomy it wants to use. It uses the `BusinessQueryManager` to find the taxonomy. The `findClassificationSchemeByName` method takes a set of `FindQualifier` objects as its first argument, but this argument can be null.

```
// Set classification scheme to NAICS
ClassificationScheme cScheme =
    bqm.findClassificationSchemeByName(null, "ntis-gov:naics");
```

The client then creates a classification using the classification scheme and a concept (a taxonomy element) within the classification scheme. For example, the following code sets up a classification for the organization within the NAICS taxonomy. The second and third arguments of the `createClassification` method are the name and value of the concept.

```
// Create and add classification
Classification classification =
    blcm.createClassification(cScheme,
        "Snack and Nonalcoholic Beverage Bars", "722213");
Collection classifications = new ArrayList();
classifications.add(classification);
org.addClassifications(classifications);
```

Services also use classifications, so you can use similar code to add a classification to a `Service` object.

Adding Services and Service Bindings to an Organization

Most organizations add themselves to a registry in order to offer services, so the JAXR API has facilities to add services and service bindings to an organization.

Like an `Organization` object, a `Service` object has a name and a description. Also like an `Organization` object, it has a unique key that is generated by the registry when the service is registered. It may also have classifications associated with it.

A service also commonly has service bindings, which provide information about how to access the service. A `ServiceBinding` object normally has a description,

an access URI, and a specification link, which provides the linkage between a service binding and a technical specification that describes how to use the service using the service binding.

The following code fragment shows how to create a collection of services, add service bindings to a service, then add the services to the organization. It specifies an access URI but not a specification link. Because the access URI is not real and because JAXR by default checks for the validity of any published URI, the binding sets its `validateURI` property to `false`.

```
// Create services and service
Collection services = new ArrayList();
Service service = blcm.createService("My Service Name");
InternationalString is =
    blcm.createInternationalString("My Service Description");
service.setDescription(is);

// Create service bindings
Collection serviceBindings = new ArrayList();
ServiceBinding binding = blcm.createServiceBinding();
is = blcm.createInternationalString("My Service Binding " +
    "Description");
binding.setDescription(is);
// allow us to publish a bogus URL without an error
binding.setValidateURI(false);
binding.setAccessURI("http://TheCoffeeBreak.com:8080/sb/");
serviceBindings.add(binding);

// Add service bindings to service
service.addServiceBindings(serviceBindings);

// Add service to services, then add services to organization
services.add(service);
org.addServices(services);
```

Saving an Organization

The primary method a client uses to add or modify organization data is the `saveOrganizations` method, which creates one or more new organizations in a registry if they did not exist previously. If one of the organizations exists but some of the data have changed, the `saveOrganizations` method updates and replaces the data.

After a client populates an organization with the information it wants to make public, it saves the organization. The registry returns the key in its response, and the client retrieves it.

```
// Add organization and submit to registry
// Retrieve key if successful
Collection orgs = new ArrayList();
orgs.add(org);
BulkResponse response = blcm.saveOrganizations(orgs);
Collection exceptions = response.getException();
if (exceptions == null) {
    System.out.println("Organization saved");

    Collection keys = response.getCollection();
    Iterator keyIter = keys.iterator();
    if (keyIter.hasNext()) {
        javax.xml.registry.infomodel.Key orgKey =
            (javax.xml.registry.infomodel.Key) keyIter.next();
        String id = orgKey.getId();
        System.out.println("Organization key is " + id);
        org.setKey(orgKey);
    }
}
```

Removing Data from the Registry

A registry allows you to remove from the registry any data that you have submitted to it. You use the key returned by the registry as an argument to one of the `BusinessLifeCycleManager` delete methods: `deleteOrganizations`, `deleteServices`, `deleteServiceBindings`, and others.

The `JAXRDelete` sample program deletes the organization created by the `JAXR-Publish` program. It deletes the organization that corresponds to a specified key string and then displays the key again so that the user can confirm that it has deleted the correct one.

```
String id = key.getId();
System.out.println("Deleting organization with id " + id);
Collection keys = new ArrayList();
keys.add(key);
BulkResponse response = blcm.deleteOrganizations(keys);
Collection exceptions = response.getException();
if (exceptions == null) {
    System.out.println("Organization deleted");
    Collection retKeys = response.getCollection();
}
```

```

        Iterator keyIter = retKeys.iterator();
        javax.xml.registry.infomodel.Key orgKey = null;
        if (keyIter.hasNext()) {
            orgKey =
                (javax.xml.registry.infomodel.Key) keyIter.next();
            id = orgKey.getId();
            System.out.println("Organization key was " + id);
        }
    }
}

```

A client can use a similar mechanism to delete services and service bindings.

Using Taxonomies in JAXR Clients

In the JAXR API, a taxonomy is represented by a `ClassificationScheme` object.

This section describes how to use the implementation of JAXR in the Sun ONE Application Server:

- To define your own taxonomies
- To specify postal addresses for an organization

Defining a Taxonomy

The JAXR specification requires a JAXR provider to be able to add user-defined taxonomies for use by JAXR clients. The mechanisms clients use to add and administer these taxonomies are implementation-specific.

The implementation of JAXR in the Sun ONE Application Server uses a simple file-based approach to provide taxonomies to the JAXR client. These files are read at run time, when the JAXR provider starts up.

The taxonomy structure for the implementation of JAXR in the Sun ONE Application Server is defined by the JAXR Predefined Concepts DTD, which is declared both in the file `jaxrconcepts.dtd` and, in XML schema form, in the file `jaxrconcepts.xsd`. The file `jaxrconcepts.xml` contains the taxonomies for the implementation of JAXR in the Sun ONE Application Server. All these files are contained in the file `<S1STUDIO_HOME>/jwsdp/common/lib/jaxr-ri.jar`. The `jaxr-ri.jar` file also includes files that define the well-known taxonomies that the implementation of JAXR in the Sun ONE Application Server uses: `naics.xml`, `iso3166.xml`, and `unspsc.xml`.

The entries in the `jaxrconcepts.xml` file look like this:

```
<PredefinedConcepts>
<JAXRClassificationScheme id="schId" name="schName">
<JAXRConcept id="schId/conCode" name="conName"
parent="parentId" code="conCode"></JAXRConcept>
...
</JAXRClassificationScheme>
</PredefinedConcepts>
```

The taxonomy structure is a containment-based structure. The element `PredefinedConcepts` is the root of the structure and must be present. The `JAXRClassificationScheme` element is the parent of the structure, and the `JAXRConcept` elements are children and grandchildren. A `JAXRConcept` element may have children, but it is not required to do so.

In all element definitions, attribute order and case are significant.

To add a user-defined taxonomy, follow these steps.

1. Publish the `JAXRClassificationScheme` element for the taxonomy as a `ClassificationScheme` object in the registry that you will be accessing. For example, you can publish the `ClassificationScheme` object to the Sun ONE Studio internal UDDI Server Registry. In order to publish a `ClassificationScheme` object, you must set its name. You also give the scheme a classification within a known classification scheme such as `uddi-org:types`. In the following code fragment, the name is the first argument of the `LifeCycleManager.createClassificationScheme` method call.

```
ClassificationScheme cScheme =
    blcm.createClassificationScheme("MyScheme",
        "A Classification Scheme");
ClassificationScheme uddiOrgTypes =
    bqm.findClassificationSchemeByName(null,
        "uddi-org:types");
if (uddiOrgTypes != null) {
    Classification classification =
        blcm.createClassification(uddiOrgTypes,
            "postalAddress", "categorization" );
    postalScheme.addClassification(classification);
    ExternalLink externalLink =
        blcm.createExternalLink(
            "http://www.mycom.com/myscheme.html",
```

```

        "My Scheme");
    postalScheme.addExternalLink(externalLink);
    Collection schemes = new ArrayList();
    schemes.add(cScheme);
    BulkResponse br =
        blcm.saveClassificationSchemes(schemes);
}

```

The BulkResponse object returned by the saveClassificationSchemes method contains the key for the classification scheme, which you need to retrieve:

```

if (br.getStatus() == JAXRResponse.STATUS_SUCCESS) {
    System.out.println("Saved ClassificationScheme");
    Collection schemeKeys = br.getCollection();
    Iterator keysIter = schemeKeys.iterator();
    while (keysIter.hasNext()) {
        javax.xml.registry.infomodel.Key key =
            (javax.xml.registry.infomodel.Key)
                keysIter.next();
        System.out.println("The postalScheme key is " +
            key.getId());
        System.out.println("Use this key as the scheme" +
            " uuid in the taxonomy file");
    }
}

```

2. In an XML file, define a taxonomy structure that is compliant with the JAXR Predefined Concepts DTD. Enter the ClassificationScheme element in your taxonomy XML file by specifying the returned key ID value as the id attribute and the name as the name attribute. For the code fragment above, for example, the opening tag for the JAXRClassificationScheme element looks something like this (all on one line):

```

<JAXRClassificationScheme
id="uuid:nnnnnnnn-nnnn-nnnn-nnnn-nnnnnnnnnnnn"
name="MyScheme">

```

The ClassificationScheme id must be a UUID.

3. Enter each JAXRConcept element in your taxonomy XML file by specifying the following four attributes, in this order:
 - a. id is the JAXRClassificationScheme id value, followed by a / separator, followed by the code of the JAXRConcept element

- b. name is the name of the JAXRConcept element
- c. parent is the immediate parent id (either the ClassificationScheme id or that of the parent JAXRConcept)
- d. code is the JAXRConcept element code value

The first JAXRConcept element in the naics.xml file looks like this (all on one line):

```
<JAXRConcept
id="uuid:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2/11"
name="Agriculture, Forestry, Fishing and Hunting"
parent="uuid:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2"
code="11"></JAXRConcept>
```

4. To add the user-defined taxonomy structure to the JAXR provider, specify the system property `com.sun.xml.registry.userTaxonomyFileNames` when you run your client program. You can use a `<sysproperty>` tag to set this property in a `build.xml` file for a client program. Or, in your program, you can set the property as follows. A vertical bar (|) is the file separator.

```
System.setProperty
("com.sun.xml.registry.userTaxonomyFileNames",
 "c:\myfile\xxx.xml|c:\myfile\xxx2.xml");
```

Specifying Postal Addresses

The JAXR specification defines a postal address as a structured interface with attributes for street, city, country, and so on. The UDDI specification, on the other hand, defines a postal address as a free-form collection of address lines, each of which may also be assigned a meaning. To map the JAXR `PostalAddress` format to a known UDDI address format, you specify the UDDI format as a `ClassificationScheme` object and then specify the semantic equivalences between the concepts in the UDDI format classification scheme and the comments in the JAXR `PostalAddress` classification scheme. The JAXR `PostalAddress` classification scheme is provided by the implementation of JAXR in the Sun ONE Application Server.

In the JAXR API, a `PostalAddress` object has the fields `streetNumber`, `street`, `city`, `state`, `postalCode` and `country`.

To specify the mapping between the JAXR postal address format and another format, you need to set two connection properties:

- The `javax.xml.registry.postalAddressScheme` property, which specifies a postal address classification scheme for the connection
- The `javax.xml.registry.semanticEquivalences` property, which specifies the semantic equivalences between the JAXR format and the other format

For example, suppose you want to use a scheme that has been published to the IBM registry and that has the UUID `uuid:6eaf4b50-4196-11d6-9e2b-000629dc0a2b`.

In the implementation of JAXR in the Sun ONE Application Server, you first use the system property `com.sun.xml.registry.userTaxonomyFileNames` to specify the concepts file where the scheme is described.

Next, you specify the postal address scheme using the `id` value from the `JAXR-ClassificationScheme` element (the UUID). Case does not matter:

```
props.setProperty("javax.xml.registry.postalAddressScheme",
    "uuid:6eaf4b50-4196-11d6-9e2b-000629dc0a2b");
```

Next, you specify the mapping from the `id` of each `JAXRConcept` element in the default JAXR postal address scheme to the `id` of its counterpart in the IBM scheme:

```
props.setProperty("javax.xml.registry.semanticEquivalences",
    "urn:uuid:PostalAddressAttributes/StreetNumber," +
    "urn:uuid:6eaf4b50-4196-11d6-9e2b-000629dc0a2b/StreetAddressNumber|" +
    "urn:uuid:PostalAddressAttributes/Street," +
    "urn:uuid:6eaf4b50-4196-11d6-9e2b-000629dc0a2b/StreetAddress|" +
    "urn:uuid:PostalAddressAttributes/City," +
    "urn:uuid:6eaf4b50-4196-11d6-9e2b-000629dc0a2b/City|" +
    "urn:uuid:PostalAddressAttributes/State," +
    "urn:uuid:6eaf4b50-4196-11d6-9e2b-000629dc0a2b/State|" +
    "urn:uuid:PostalAddressAttributes/PostalCode," +
    "urn:uuid:6eaf4b50-4196-11d6-9e2b-000629dc0a2b/ZipCode|" +
    "urn:uuid:PostalAddressAttributes/Country," +
    "urn:uuid:6eaf4b50-4196-11d6-9e2b-000629dc0a2b/Country");
```

After you create the connection using these properties, you can create a postal address and assign it to the primary contact of the organization before you publish the organization:

```
String streetNumber = "99";
String street = "Imaginary Ave. Suite 33";
String city = "Imaginary City";
String state = "NY";
String country = "USA";
String postalCode = "00000";
String type = "";
PostalAddress postAddr =
    blcm.createPostalAddress(streetNumber, street, city, state,
        country, postalCode, type);
Collection postalAddresses = new ArrayList();
postalAddresses.add(postAddr);
primaryContact.setPostalAddresses(postalAddresses);
```

A JAXR query can then retrieve the postal address using `PostalAddress` methods, if the postal address scheme and semantic equivalences for the query are the same as those specified for the publication. To retrieve postal addresses when you do not know what postal address scheme was used to publish them, you can retrieve them as a collection of `Slot` objects. The `JAXRQueryPostal.java` sample program shows how to do this.

In general, you can create a user-defined postal address taxonomy for any `postalAddress` tModels that use the well-known categorization in the `uddi-org:types` taxonomy, which has the tModel UUID `uuid:c1acf26d-9672-4404-9d70-39b756e62ab4` with a value of `postalAddress`. You can retrieve the tModel `overviewDoc`, which points to the technical detail for the specification of the scheme, where the taxonomy structure definition can be found. (The JAXR equivalent of an `overviewDoc` is an `ExternalLink`.)

Running the Client Examples

The simple client programs provided with this tutorial can be run from the command line. You can modify them to suit your needs. They allow you to specify the IBM registry, the Microsoft registry, or the Registry Server for queries and updates; you can specify any other UDDI version 2 registry.

The client examples, in the `<INSTALL>/j2eetutorial/examples/jaxr` directory, are as follows:

- `JAXRQuery.java` shows how to search a registry for organizations
- `JAXRQueryByNAICSClassification.java` shows how to search a registry using a common classification scheme
- `JAXRQueryByWSDLClassification.java` shows how to search a registry for Web services that describe themselves by means of a WSDL document
- `JAXRPublish.java` shows how to publish an organization to a registry
- `JXRDelete.java` shows how to remove an organization from a registry
- `JAXRSaveClassificationScheme.java` shows how to publish a classification scheme (specifically, a postal address scheme) to a registry
- `JAXRPublishPostal.java` shows how to publish an organization with a postal address for its primary contact
- `JAXRQueryPostal.java` shows how to retrieve postal address data from an organization
- `JXRDeleteScheme.java` shows how to delete a classification scheme from a registry
- `JAXRGetMyObjects.java` lists all the objects that you own in a registry

The `<INSTALL>/j2eetutorial/examples/jaxr` directory also contains:

- A `JAXRExamples.properties` file, in the `src` subdirectory, that supplies string values used by the sample programs
- A file called `postalconcepts.xml` that you use with the postal address examples
- A file called `postal.properties` that specifies the location of the user taxonomy file for the postal address examples

The instructions that follow assume that you have started Sun ONE Studio. You do not need to have the Sun ONE Application Server running in order to use JAXR.

Mounting the Filesystem

Before you can compile the examples, you must mount the filesystem. To do so:

1. Select the Filesystems tab of the Explorer.
2. Choose Mount Filesystem from the File menu.

3. Mount the following directory:

`<INSTALL>/j2eetutorial/examples/jaxr`

Editing the Properties File

Before you compile the examples, edit the file `JAXRExamples.properties` as follows.

1. Right-click the file and choose the Edit menu item.
2. Edit the following lines to specify the registry you wish to access. For both the `queryURL` and the `publishURL` assignments, comment out all but the registry you wish to access. The default is the Sun ONE Studio internal UDDI Server Registry, so if you will be using this registry on your own system, you do not need to change this section.

```
## Uncomment one pair of query and publish URLs.  
## IBM:  
#query.url=http://uddi.ibm.com/testregistry/inquiryapi  
#publish.url=https://uddi.ibm.com/testregistry/protect/  
publishapi  
## Microsoft:  
#query.url=http://test.uddi.microsoft.com/inquire  
#publish.url=https://test.uddi.microsoft.com/publish  
## Registry Server:  
query.url=http://localhost:8089/registry-server/  
RegistryServerServlet  
publish.url=http://localhost:8089/registry-server/  
RegistryServerServlet
```

If the internal UDDI Server Registry is running on a system other than your own, specify the fully qualified host name instead of `localhost`. Do not use `https:` for the `publishURL`.

The IBM and Microsoft registries both have a considerable amount of data in them that you can perform queries on. Moreover, you do not have to register if you are only going to perform queries.

We have not included the URLs of the SAP registry; feel free to add them.

If you want to publish to any of the public registries, the registration process for obtaining access to them is not difficult (see Preliminaries: Getting Access to a Registry, page 351). Each of them, however, allows you

to have only one organization registered at a time. If you publish an organization to one of them, you must delete it before you can publish another. Since the organization that the JAXRPublish example publishes is fictitious, you will want to delete it immediately anyway.

The internal UDDI Server Registry gives you more freedom to experiment with JAXR. You can publish as many organizations to it as you wish. However, this registry comes with an empty database, so you must publish organizations to it yourself before you can perform queries on the data.

3. Edit the following lines to specify the user name and password you obtained when you registered with the registry. The default is the internal UDDI Server Registry default password.

```
## Specify username and password if needed
## testuser/testuser are defaults for internal Registry
registry.username=testuser
registry.password=testuser
```

4. If you will be using a public registry, edit the following lines, which contain empty strings for the proxy hosts, to specify your own proxy settings. The proxy host is the system on your network through which you access the Internet; you usually specify it in your Internet browser settings. You can leave this value empty to use the internal UDDI Server Registry.

```
## HTTP and HTTPS proxy host and port;
## ignored by internal Registry
http.proxyHost=
http.proxyPort=8080
https.proxyHost=
https.proxyPort=8080
```

The proxy ports have the value 8080, which is the usual one; change this string if your proxy uses a different port.

For a public registry, your entries usually follow this pattern:

```
http.proxyHost=proxyhost.mydomain
http.proxyPort=8080
https.proxyHost=proxyhost.mydomain
https.proxyPort=8080
```

5. Feel free to change any of the organization data in the remainder of the file. This data is used by the publishing examples, JAXRPublish and JAXRPublishPostal.

6. Save and close the file.

You can edit the `JAXRExamples.properties` file at any time. When you run the client examples, they use the latest version of the file.

Starting the Internal UDDI Server Registry

If you plan to use the internal UDDI Server Registry, perform the following steps:

1. Select the Runtime tab of the Explorer.
2. Expand the UDDI Server Registry node.
3. Right-click the Internal UDDI Registry node and choose the Start Server menu item.

Setting the Compilation Classpath

Before you can compile the programs, you need to set the compilation classpath. To do so:

1. Choose Options from the Tools menu.
2. Expand the Building node, then the Compiler Types node.
3. Choose External Compilation.
4. Select the Expert tab.
5. Click the Class Path property, then double-click the ellipsis in the value field.
6. In the property editor, click Add JAR/Zip.
7. In the file chooser, navigate to the directory `<S1STUDIO_HOME>/jwsdp/common/lib` and choose the `jaxr-api.jar` file. (This file is also in `<S1AS7_HOME>/share/lib`.)
8. Click OK.
9. Click OK in the property editor, then click Close in the Options window.

Compiling the Examples

To compile the programs:

1. In the Filesystems tab of the Explorer, right-click the `<INSTALL>/j2eetutorial/examples/jaxr` directory.
2. Choose the Compile All menu item.

Compiler messages appear in an output window.

Running the JAXRPublish Example

To run the JAXRPublish program, right-click the file in the Filesystems tab and choose the Execute menu item.

The program output appears in an output window associated with the Running tab of the main window. It displays the string value of the key of the new organization, which is named “The Coffee Break.”

Do not dismiss the output window. You’ll need its contents in the future. Return to the Editing tab to continue running the examples.

After you run the JAXRPublish program but before you run JAXRDelete, you can run JAXRQuery to look up the organization you published.

Running the JAXRQuery Example

To run the JAXRQuery example:

1. Click the JAXRQuery file in the Filesystems tab.
2. In the property window for the file, choose the Execution tab.
3. Click Arguments.
4. Enter a string in the value field, such as `coffee`.
5. Right-click the JAXRQuery file in the Filesystems tab and choose the Execute menu item.

The program searches the registry for organizations whose names contain the string you specified. Searching is not case-sensitive.

Running the JAXRQueryByNAICSCClassification Example

After you run the JAXR Publish program, you can also run the JAXRQueryByNAICSCClassification example, which looks for organizations that use the “Snack and Nonalcoholic Beverage Bars” classification, the same one used for the organization created by JAXR Publish.

To run the program, right-click the JAXRQueryByNAICSCClassification file in the Filesystems tab and choose the Execute menu item

Running the JAXRDelete Example

To run the JAXRDelete example, you need to provide the key returned by the JAXR Publish program. Perform the following steps:

1. Click the Running tab in the main window.
2. In the Output window tab for the JAXR Publish example, select the organization key value from the program output. It looks something like f2c2827a-b1f2-c282-3fd27a5c1893. Right-click in the window and choose the Copy menu item.
3. Click the JAXRDelete file in the Filesystems tab.
4. In the property window for the file, choose the Execution tab.
5. Click Arguments.
6. Paste the key string you copied in step 2 into the value field.
7. Right-click the JAXRDelete file in the Filesystems tab and choose the Execute menu item.

The program deletes the specified organization from the registry.

Running the JAXRQueryByWSDLCClassification Example

To run the JAXRQueryByWSDLCClassification example, right-click the file in the Filesystems tab and choose the Execute menu item.

This example returns many results from the public registries and is likely to run for several minutes.

Publishing a Classification Scheme

In order to publish organizations with postal addresses to public registries, you must publish a classification scheme for the postal address first.

To run the `JAXRSaveClassificationScheme` program, right-click the file in the Filesystems tab and choose the Execute menu item.

The program returns a UUID string, which you will use in the next section.

The public registries allow you to own more than one classification scheme at a time (the limit is usually a total of about 10 classification schemes and concepts put together).

Running the Postal Address Examples

Running the postal address examples involves the following steps:

1. Specifying the UUID of the classification scheme in the `postalconcepts.xml` file
2. Specifying the pathname of the `postalconcepts.xml` file in the `postal.properties` file
3. Running the `JAXRPublishPostal` example
4. Running the `JAXRQueryPostal` example

Specifying the UUID

Before you run the postal address examples, you need to specify the UUID of the classification scheme in the `postalconcepts.xml` file:

1. Click the Running tab in the main window.
2. In the Output window tab for the `JAXRSaveClassificationScheme` example, select the postal scheme key value from the program output. It looks something like this: `uuid:f2be7262-aff2-be76-b3fa-ddd3e4e600f5`. Right-click in the window and choose the Copy menu item.
3. Right-click the file `postalconcepts.xml` and choose the Edit menu item.

4. Wherever you see the string `uuid-from-save`, replace it with the UUID string you copied in step 2.
5. Save and close the file.

For a given registry, you only need to save the classification scheme and edit `postalconcepts.xml` once. After you perform those steps, you can run the `JAXRPublishPostal` and `JAXRQueryPostal` programs multiple times.

Editing the Properties File

Next, edit the `postal.properties` file to specify the correct pathname for the `postalconcepts.xml` file:

1. Right-click the file `postal.properties` and choose the Edit menu item.
2. Specify the correct pathname for the `postalconcepts.xml` file. On a UNIX system the pathname may be relative, but on a Windows system it must be absolute and must be specified using double backslashes. For example, you might enter the following (all on one line):

```
com.sun.xml.registry.userTaxonomyFileNames=D:\\Sun\\AppServer7\\docs\\tutorial\\examples\\jaxr\\postalconcepts.xml
```

Running the JAXRPublishPostal Example

The `JAXRPublishPostal` example uses the `postal.properties` file to set the system property `com.sun.xml.registry.userTaxonomyFileNames` to the correct location of the `postalconcepts.xml` file. You need to specify this file and the UUID string of the classification scheme as command-line arguments.

1. Click the `JAXRPublishPostal` file in the Filesystems tab.
2. In the property window for the file, choose the Execution tab.
3. Click Arguments, then double-click the ellipsis to bring up the property editor.
4. Enter the pathname of the `postal.properties` file and the UUID string of the classification scheme. On a Windows system, you must specify the full pathname. For example, you might specify the following, all on one line:

```
D:\\MyTutorial\\j2eetutorial\\examples\\jaxr\\postal.properties uuid:f2be7262-aff2-be76-b3fa-ddd3e4e600f5
```

5. Click OK. Notice that on a Windows system, the pathname appears with double backslashes.
6. Right-click the JAXRPublishPostal file and choose the Execute menu item.

The program output displays the string value of the key of the new organization.

Running the JAXRQueryPostal Example

Like the JAXRPublishPostal example, the JAXRQueryPostal example expects the postal.properties file pathname and the UUID string as command-line arguments. It also expects a query string.

1. Click the JAXRQueryPostal file in the Filesystems tab.
2. In the property window for the file, choose the Execution tab.
3. Click Arguments, then double-click the ellipsis to bring up the property editor.
4. Enter the pathname of the postal.properties file, the query string, and the UUID string of the classification scheme. On a Windows system, you must specify the full pathname. For example, you might specify the following, all on one line:

```
D:\\Sun\\AppServer7\\docs\\tutorial\\examples\\jaxr\\  
postal.properties coffee uuid:f2be7262-aff2-be76-b3fa-  
ddd3e4e600f5
```

5. Right-click the JAXRQueryPostal file and choose the Execute menu item.
The postal address for the primary contact will appear correctly with the JAXR PostalAddress methods. Any postal addresses found that use other postal address schemes will appear as Slot lines.
6. If you are using a public registry, make sure to follow the instructions in Running the JAXRDelete Example (page 377) to delete the organization you published.

Deleting a Classification Scheme

You may or may not want to delete the classification scheme you published. For a UDDI registry, deleting a classification scheme removes it from the registry logically but not physically. You can no longer use the classification scheme, but it will still be visible if, for example, you call the method QueryMan-

`ager.getRegisteredObjects`. Since the public registries allow you to own up to 10 of these objects, this is not likely to be a problem. However, once you have created a classification scheme for postal addresses in a public registry, you may want to leave it there for future use.

The Sun ONE Studio internal UDDI Server Registry imposes no limit on the number of classification schemes you can own.

To delete the classification scheme you published after you have finished using it, run the `JAXRDeleteScheme` program as follows:

1. Click the Running tab in the main window.
2. In the Output window tab for either the `JAXRSaveClassificationScheme`, `JAXRPublishPostal`, or `JAXRQueryPostal` example, select the value of the UUID string from the program output. Right-click in the window and choose the Copy menu item.
3. Click the `JAXRDeleteScheme` file in the Filesystems tab.
4. In the property window for the file, choose the Execution tab.
5. Click Arguments.
6. Paste the UUID string you copied in step 2 into the value field.
7. Right-click the `JAXRDeleteScheme` file in the Filesystems tab and choose the Execute menu item.

The program deletes the specified classification scheme from the registry.

Getting a List of Your Registry Objects

To get a list of the objects you own in the registry, both organizations and classification schemes, run the `JAXRGetMyObjects` program.

Right-click the file in the Filesystems tab and choose the Execute menu item.

If you run this program with the Registry Server, it returns all the standard UDDI taxonomies provided with the Registry Server, not just the objects you have created.

Stopping the Internal UDDI Server Registry

If you started the internal UDDI Server Registry, perform the following steps after you have finished using the examples:

1. Select the Runtime tab of the Explorer.
2. Expand the UDDI Server Registry node.
3. Right-click the Internal UDDI Registry node and choose the Stop Server menu item.

Further Information

For more information about JAXR, registries, and Web services, see the following:

- Java Specification Request (JSR) 93: JAXR 1.0:
<http://jcp.org/jsr/detail/093.jsp>
- JAXR home page:
<http://java.sun.com/xml/jaxr/index.html>
- Universal Description, Discovery, and Integration (UDDI) project:
<http://www.uddi.org/>
- ebXML:
<http://www.ebxml.org/>
- Open Source JAXR Provider for ebXML Registries:
https://sourceforge.net/forum/forum.php?forum_id=197238
- Java Technology and XML:
<http://java.sun.com/xml/>
- Java Technology & Web Services:
<http://java.sun.com/webservices/index.html>

The Coffee Break Application

Stephanie Bodoff, Maydene Fisher, Dale Green, Kim Haase

The introduction to this tutorial introduced a scenario in which an application (The Coffee Break) is constructed using Web services. Now that we have discussed all the technologies necessary to build Web applications and Web services, this chapter describes an implementation of the scenario described in Chapter 8.

Coffee Break Overview

The Coffee Break sells coffee on the Internet. Customers communicate with the Coffee Break server to order coffee online. The server consists of Java Servlets, JSP pages, and JavaBeans components. A customer enters the quantity of each coffee to order and clicks the “Submit” button to send the order.

The Coffee Break does not maintain any inventory. It handles customer and order management and billing. Each order is filled by forwarding suborders to one or more coffee distributors. This process is depicted in Figure 12–1.

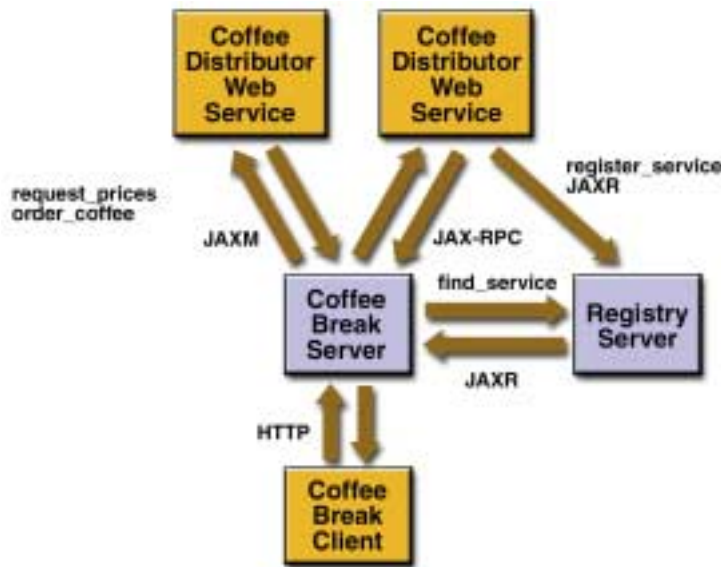


Figure 12-1 Coffee Break Application Flow

The Coffee Break server obtains the coffee varieties it sells and their prices by querying distributors at startup and on demand.

1. The Coffee Break server uses JAXM messaging to communicate with one of its distributors. It has been dealing with this distributor for some time and has previously made the necessary arrangements for doing request-response JAXM messaging. The two parties have agreed to exchange four kinds of XML messages and have set up the DTDs those messages will follow.
2. The Coffee Break server uses JAXR to send a query searching for coffee distributors that support JAX-RPC to the Registry Server.
3. The Coffee Break server requests price lists from each of the coffee distributors. The server makes the appropriate remote procedure calls and waits for the response, which is a JavaBeans component representing a price list. The JAXM distributor returns price lists as XML documents.
4. Upon receiving the responses, the Coffee Break server processes the price lists from the JavaBeans components returned by calls to the distributors.
5. The Coffee Break Server creates a local database of distributors.
6. When an order is placed, suborders are sent to one or more distributors using the distributor's preferred protocol.

JAX-RPC Distributor Service

The Coffee Break server is a client of the JAX-RPC distributor service. To examine the code for the service, in the IDE mount the `<INSTALL>/j2eetutorial/examples/cb/jaxrpc` directory. The service supports two operations: fetching the price list and placing an order.

Fetching the Price List

This operation is implemented by the `getPriceList` method of the `jaxrpc.service.Supplier`. The `getPriceList` method returns a `PriceListBean` object, which lists the name and price of each type of coffee that can be ordered from this service. The `getPriceList` method creates the `PriceListBean` object by invoking a private method named `loadPrices`. In a production application, the `loadPrices` method would fetch the prices from a database. However, our `loadPrices` method takes a shortcut by invoking `PriceLoader.loadItems`, a method with hardcoded prices. Here are the `getPriceList` and `loadPrices` methods:

```
public PriceListBean getPriceList() {  
    PriceListBean priceList = loadPrices();  
    return priceList;  
}  
  
private PriceListBean loadPrices() {  
    Date today = new Date();  
    Date endDate = DateHelper.addDays(today, 30);  
    Collection priceItems = PriceLoader.loadItems();  
    PriceListBean priceList = new PriceListBean();  
    priceList.setStartDate(today);  
    priceList.setEndDate(endDate);  
    priceList.setPriceItems(priceItems);  
    return priceList;  
}
```

Placing an Order

The `placeOrder` method of `jaxrpc.service.Supplier` accepts as input a coffee order and returns a confirmation for the order. To keep things simple, the `placeOrder` method confirms every order and sets the ship date in the confirmation to the next day. (This date is calculated by `DateHelper`, a utility class that resides in the `jaxrpc.service` package.) The source code for the `placeOrder` method follows:

```
public ConfirmationBean placeOrder(OrderBean order) {  
  
    Date tomorrow =  
        com.sun.cb.DateHelper.addDays(new Date(), 1);  
    ConfirmationBean confirmation =  
        new ConfirmationBean(order.getId(), tomorrow);  
    return confirmation;  
}
```

Publishing the Service in the Registry

Because we want customers to find our service, we will publish it in a registry. The programs that publish and remove our service are called `OrgPublisher` and `OrgRemover`. Although related to the service, these programs are not part of the service's Web application (that is, the servlet that implements the service). They are stand-alone programs that are run separately. These programs reside in the `<INSTALL>/j2eetutorial/examples/cb/jaxrpc/registry` directory.

The `OrgPublisher` program begins by loading `String` values from the `CoffeeRegistry.properties` file. Next, the program instantiates a utility class named `JAXRPublisher`. `OrgPublisher` connects to the registry by invoking the `makeConnection` method of `JAXRPublisher`. To publish the service, `OrgPublisher` invokes the `executePublish` method, which accepts as input `username`, `password`, and `endpoint`. The `username` and `password` values are required by the Registry Server. The `endpoint` value is the URL that remote clients will use to contact our JAX-RPC service. The `executePublish` method of `JAXRPublisher` returns a key that uniquely identifies the service in the registry. `OrgPublisher` saves this key in a text file named `orgkey.txt`. The `OrgRemover` program will read the key from `orgkey.txt` so that it can delete the service. (See

Deleting the Service From the Registry, page 391.) The source code for the OrgPublisher program follows.

```
package jaxrpc.registry;

import javax.xml.registry.*;
import java.util.ResourceBundle;
import java.io.*;

public class OrgPublisher {

    public static void main(String[] args) {

        ResourceBundle registryBundle =
            ResourceBundle.getBundle
                ("jaxrpc.registry.CoffeeRegistry");

        String queryURL =
            registryBundle.getString("query.url");
        String publishURL =
            registryBundle.getString("publish.url");
        String username =
            registryBundle.getString("registry.username");
        String password =
            registryBundle.getString("registry.password");
        String endpoint = registryBundle.getString("endpoint");
        String keyFile = registryBundle.getString("key.file");

        JAXRPublisher publisher = new JAXRPublisher();
        publisher.makeConnection(queryURL, publishURL);
        String key = publisher.executePublish
            (username, password, endpoint);

        try {
            FileWriter out = new FileWriter(keyFile);
            out.write(key);
            out.flush();
            out.close();
        } catch (IOException ex) {
            System.out.println(ex.getMessage());
        }

    }

}
```

The JAXRPublisher class is almost identical to the sample program JAXRPublisher.java, which is described in Managing Registry Data (page 361).

First, the `makeConnection` method creates a connection to the Registry Server. See *Establishing a Connection* (page 351) for more information. To do this, it first specifies a set of connection properties using the query and publish URLs passed in from the `CoffeeRegistry.properties` file. For the Registry Server, the query and publish URLs are actually the same.

```
Properties props = new Properties();
props.setProperty("javax.xml.registry.queryManagerURL",
    queryUrl);
props.setProperty("javax.xml.registry.lifeCycleManagerURL",
    publishUrl);
```

Next, the `makeConnection` method creates the connection, using the connection properties:

```
ConnectionFactory factory = ConnectionFactory.newInstance();
factory.setProperties(props);
connection = factory.createConnection();
```

The `executePublish` method takes three arguments: a username, a password, and an endpoint. It begins by obtaining a `RegistryService` object, then a `BusinessQueryManager` object and a `BusinessLifeCycleManager` object, which enable it to perform queries and manage data:

```
rs = connection.getRegistryService();
blcm = rs.getBusinessLifeCycleManager();
bqm = rs.getBusinessQueryManager();
```

Because it needs password authentication in order to publish data, it then uses the username and password arguments to establish its security credentials:

```
PasswordAuthentication passwdAuth =
    new PasswordAuthentication(username,
        password.toCharArray());
Set creds = new HashSet();
creds.add(passwdAuth);
connection.setCredentials(creds);
```

It then creates an `Organization` object with the name “JAXRPCCoffeeDistributor,” then a `User` object that will serve as the primary contact. It gets the data

from the resource bundle instead of hardcoding it as strings, but otherwise this code is almost identical to that shown in the JAXR chapter.

```
ResourceBundle bundle =
    ResourceBundle.getBundle("jaxrpc.registry.CoffeeRegistry");

// Create organization name and description
Organization org =
    blcm.createOrganization(bundle.getString("org.name"));
InternationalString s =
    blcm.createInternationalString
        (bundle.getString("org.description"));
org.setDescription(s);

// Create primary contact, set name
User primaryContact = blcm.createUser();
PersonName pName =
    blcm.createPersonName(bundle.getString("person.name"));
primaryContact.setPersonName(pName);
```

It adds a telephone number and email address for the user, then makes the user the primary contact:

```
org.setPrimaryContact(primaryContact);
```

It gives JAXRPPCoffeeDistributor a classification using the North American Industry Classification System (NAICS). In this case it uses the classification “Other Grocery and Related Products Wholesalers”.

```
Classification classification = (Classification)
    blcm.createClassification(cScheme,
        bundle.getString("classification.name"),
        bundle.getString("classification.value"));
Collection classifications = new ArrayList();
classifications.add(classification);
org.addClassifications(classifications);
```

Next, it adds the JAX-RPC service, called “JAXRPPCoffee Service,” and its service binding. The access URI for the service binding contains the endpoint URL that remote clients will use to contact our service:

```
http://localhost:80/SupplierService/SupplierService
```

JAXR validates each URI, so an exception is thrown if the service was not deployed before you ran this program.

```

Collection services = new ArrayList();
Service service =
    blcm.createService(bundle.getString("service.name"));
InternationalString is =
    blcm.createInternationalString
        (bundle.getString("service.description"));
service.setDescription(is);

// Create service bindings
Collection serviceBindings = new ArrayList();
ServiceBinding binding = blcm.createServiceBinding();
is = blcm.createInternationalString
    (bundle.getString("service.binding"));
binding.setDescription(is);
try {
    binding.setAccessURI(endpoint);
} catch (JAXRException je) {
    throw new JAXRException("Error: Publishing this " +
        "service in the registry has failed because " +
        "the service has not been deployed on the application
server.");
}
serviceBindings.add(binding);

// Add service bindings to service
service.addServiceBindings(serviceBindings);

// Add service to services, then add services to organization
services.add(service);
org.addServices(services);

```

Then it saves the organization to the registry:

```

Collection orgs = new ArrayList();
orgs.add(org);
BulkResponse response = blcm.saveOrganizations(orgs);

```

The BulkResponse object returned by saveOrganizations includes the Key object containing the unique key value for the organization. The executePublish method first checks to make sure the saveOrganizations call succeeded.

If the call succeeded, the method extracts the value from the Key object and displays it:

```
Collection keys = response.getCollection();
Iterator keyIter = keys.iterator();
if (keyIter.hasNext()) {
    javax.xml.registry.infomodel.Key orgKey =
        (javax.xml.registry.infomodel.Key) keyIter.next();
    id = orgKey.getId();
    System.out.println("Organization key is " + id);
}
```

Finally, the method returns the string `id` so that the `OrgPublisher` program can save it in a file for use by the `OrgRemover` program.

Deleting the Service From the Registry

The `OrgRemover` program deletes the service from the Registry Server immediately before the service is removed. Like the `OrgPublisher` program, the `OrgRemover` program starts by fetching values from the `CoffeeRegistry.properties` file. One these values, `keyFile`, is the name of the file that contains the key that uniquely identifies the service. `OrgPublisher` reads the key from the file, connects to the Registry Server by invoking `makeConnection`, and then deletes the service from the registry by calling `executeRemove`. Here is the source code for the `OrgRemover` program:

```
package jaxrpc.registry;

import java.util.ResourceBundle;
import javax.xml.registry.*;
import javax.xml.registry.infomodel.Key;
import java.io.*;

public class OrgRemover {

    Connection connection = null;

    public static void main(String[] args) {

        String keyStr = null;

        ResourceBundle registryBundle =
            ResourceBundle.getBundle
                ("jaxrpc.registry.CoffeeRegistry");
```

```

String queryURL =
    registryBundle.getString("query.url");
String publishURL =
    registryBundle.getString("publish.url");
String username =
    registryBundle.getString("registry.username");
String password =
    registryBundle.getString("registry.password");
String keyFile = registryBundle.getString("key.file");

try {
    FileReader in = new FileReader(keyFile);
    char[] buf = new char[512];
    while (in.read(buf, 0, 512) >= 0) {
        in.close();
        keyStr = new String(buf).trim();
    } catch (IOException ex) {
        System.out.println(ex.getMessage());
    }

    JAXRRemover remover = new JAXRRemover();
    remover.makeConnection(queryURL, publishURL);
    javax.xml.registry.infomodel.Key modelKey = null;
    modelKey = remover.createOrgKey(keyStr);
    remover.executeRemove(modelKey, username, password);
}
}

```

Instantiated by the `OrgRemover` program, the `JAXRRemover` class contains the `makeConnection`, `createOrgKey`, and `executeRemove` methods. It is almost identical to the sample program `JAXRDelete.java`, which is described in *Removing Data from the Registry* (page 365).

The `makeConnection` method is identical to the `JAXRPublisher` method of the same name.

The `createOrgKey` method is a utility method that takes one argument, the string value extracted from the key file. It obtains the `RegistryService` object and the `BusinessLifecycleManager` object, then creates a `Key` object from the string value.

The `executeRemove` method takes three arguments: a username, a password, and the `Key` object returned by the `createOrgKey` method. It uses the username and password arguments to establish its security credentials with the Registry Server, just as the `executePublish` method does.

The method then wraps the Key object in a Collection and uses the BusinessLifeCycleManager object's deleteOrganizations method to delete the organization.

```
Collection keys = new ArrayList();
keys.add(key);
BulkResponse response = blcm.deleteOrganizations(keys);
```

The deleteOrganizations method returns the keys of the organizations it deleted, so the executeRemove method then verifies that the correct operation was performed and displays the key for the deleted organization.

```
Collection retKeys = response.getCollection();
Iterator keyIter = retKeys.iterator();
javax.xml.registry.infomodel.Key orgKey = null;
if (keyIter.hasNext()) {
    orgKey =
        (javax.xml.registry.infomodel.Key) keyIter.next();
    id = orgKey.getId();
    System.out.println("Organization key was " + id);
}
```

JAXM Distributor Service

The JAXM distributor service and the Coffee Break have made arrangements regarding their exchange of XML documents. These arrangements include what kinds of messages they will send, the form of those messages, and what kind of JAXM messaging they will do. If they had agreed to do one-way messaging, they would also have had to use messaging providers that talk to each other and had to use the same profile. In this scenario, the parties have agreed to use request-response messaging, so a messaging provider is not needed.

The Coffee Break server sends two kinds of messages:

- Requests for current wholesale coffee prices
- Customer orders for coffee

The JAXM coffee supplier responds with two kinds of messages:

- Current price lists
- Order confirmations

All of the messages they send conform to an agreed-upon XML structure, which is specified in a DTD for each kind of message. This allows them to exchange messages even though they use different document formats internally.

The four kinds of messages exchanged by the Coffee Break server and the JAXM distributor are specified by the following DTDs:

- request-prices.dtd
- price-list.dtd
- coffee-order.dtd
- confirm.dtd

These DTDs may be found at

`<INSTALL>/j2eetutorial/examples/cb/jaxm/dtds`

The dtds directory also contains a sample of what the XML documents specified in the DTDs might look like. The corresponding XML files for each of the DTDs are as follows:

- request-prices.xml
- price-list.xml
- coffee-order.xml
- confirm.xml

Because of the DTDs, both parties know ahead of time what to expect in a particular kind of message and can therefore extract its content using the JAXM API.

Code for the server application is in the directory:

`<INSTALL>/j2eetutorial/examples/cb/jaxm/service`

JAXM Service

The JAXM coffee distributor, the JAXM server in this scenario, provides the response part of the request-response paradigm. When JAXM messaging is being used, the server code is a servlet. The core part of each servlet is made up of three `javax.servlet.HttpServlet` methods: `init`, `doPost`, and `onMessage`. The `init` and `doPost` methods set up the response message, and the `onMessage` method gives the message its content.

Returning the Price List

This section takes you through the servlet `PriceListServlet`. This servlet creates the message with the current price list that is returned to the method call, invoked in `PriceListRequest`.

Any servlet extends a `javax.servlet` class. Being part of a Web application, this servlet extends `HttpServlet`. It first creates a static `MessageFactory` object that will be used later to create the `SOAPMessage` object that is returned. Then it declares the `MessageFactory` object `msgFactory`, which will be used to create a `SOAPMessage` object that has the headers and content of the original request message.

```
public class PriceListServlet extends HttpServlet {  
    MessageFactory msgFactory;
```

Every servlet has an `init` method. This `init` method initializes the servlet with the configuration information that the application server passed to it. Then it simply initializes `msgFactory` with the default implementation of the `MessageFactory` class.

```
    public void init(ServletConfig servletConfig)  
        throws ServletException {  
        super.init(servletConfig);  
        try {  
            // Initialize it to the default.  
            msgFactory = MessageFactory.newInstance();  
        } catch (SOAPException ex) {  
            throw new ServletException(  
                "Unable to create message factory" + ex.getMessage());  
        }  
    }  
}
```

The next method defined in `PriceListServlet` is `doPost`, which does the real work of the servlet by calling the `onMessage` method. (The `onMessage` method is discussed later in this section.) the application server passes the `doPost` method two arguments. The first argument, the `HttpServletRequest` object `req`, holds the content of the message sent in `PriceListRequest`. The `doPost` method gets the content from `req` and puts it in the `SOAPMessage` object `msg` so that it can pass it to the `onMessage` method. The second argument, the `HttpServletResponse` object `resp`, will hold the message generated by executing the method `onMessage`.

In the following code fragment, `doPost` calls the methods `getHeaders` and `putHeaders`, defined immediately after `doPost`, to read and write the headers in `req`. It then gets the content of `req` as a stream and passes the headers and the input stream to the method `MessageFactory.createMessage`. The result is that the `SOAPMessage` object `msg` contains the request for a price list. Note that in this case, `msg` does not have any headers because the message sent in `PriceListRequest` did not have any headers.

```
public void doPost( HttpServletRequest req, HttpServletResponse
    resp) throws ServletException, IOException {
    try {
        // Get all the headers from the HTTP request.
        MimeHeaders headers = getHeaders(req);

        // Get the body of the HTTP request.
        InputStream is = req.getInputStream();

        // Now internalize the contents of the HTTP request and
        // create a SOAPMessage
        SOAPMessage msg = msgFactory.createMessage(headers, is);
```

Next, the code declares the `SOAPMessage` object `reply` and populates it by calling the method `onMessage`.

```
        SOAPMessage reply = null;
        reply = onMessage(msg);
```

If `reply` has anything in it, its contents are saved, the status of `resp` is set to `OK`, and the headers and content of `reply` are written to `resp`. If `reply` is empty, the status of `resp` is set to indicate that there is no content.

```
        if (reply != null) {
            // Need to call saveChanges because we're going to use the
            // MimeHeaders to set HTTP response information. These
            // MimeHeaders are generated as part of the save.
```



```

        if (reply.saveRequired()) {
            reply.saveChanges();
        }

        resp.setStatus(HttpServletResponse.SC_OK);

        putHeaders(reply.getMimeHeaders(), resp);
        // Write out the message on the response stream.
        OutputStream os = resp.getOutputStream();
        reply.writeTo(os);
        os.flush();
    } else
        resp.setStatus(HttpServletResponse.SC_NO_CONTENT);

    } catch (Exception ex) {
        throw new ServletException( "JAXM POST failed " +
            ex.getMessage());
    }
}

```

The methods `getHeaders` and `putHeaders` are not standard methods in a servlet the way `init`, `doPost`, and `onMessage` are. The method `doPost` calls `getHeaders` and passes it the `HttpServletRequest` object `req` that the application server passed to it. It returns a `MimeHeaders` object populated with the headers from `req`.

```

static MimeHeaders getHeaders(HttpServletRequest req) {

    Enumeration enum = req.getHeaderNames();
    MimeHeaders headers = new MimeHeaders();

    while (enum.hasMoreElements()) {
        String headerName = (String)enum.nextElement();
        String headerValue = req.getHeader(headerName);

        StringTokenizer values = new StringTokenizer(
            headerValue, ",");
        while (values.hasMoreTokens()) {
            headers.addHeader(headerName,
                values.nextToken().trim());
        }
    }

    return headers;
}

```

The `doPost` method calls `putHeaders` and passes it the `MimeHeaders` object `headers`, which was returned by the method `getHeaders`. The method `putHeaders` writes the headers in `headers` to `res`, the second argument passed to it. The result is that `res`, the response that the application server will return to the method call, now contains the headers that were in the original request.

```
static void putHeaders(MimeHeaders headers,
                      HttpServletResponse res) {
    Iterator it = headers.getAllHeaders();
    while (it.hasNext()) {
        MimeHeader header = (MimeHeader)it.next();
        String[] values = headers.getHeader(header.getName());
        if (values.length == 1)
            res.setHeader(header.getName(),
                          header.getValue());
        else {
            StringBuffer concat = new StringBuffer();
            int i = 0;
            while (i < values.length) {
                if (i != 0) concat.append(',');
                concat.append(values[i++]);
            }
            res.setHeader(header.getName(), concat.toString());
        }
    }
}
```

The method `onMessage` is the application code for responding to the message sent by `PriceListRequest` and internalized into `msg`. It uses the static `MessageFactory` object `fac` to create the `SOAPMessage` object `message` and then populates it with the distributor's current coffee prices.

The method `doPost` invokes `onMessage` and passes it `msg`. In this case, `onMessage` does not need to use `msg` because it simply creates a message containing the distributor's price list. The `onMessage` method in `ConfirmationServlet` (Returning the Order Confirmation, page 400), on the other hand, uses the message passed to it to get the order ID.

```
public SOAPMessage onMessage(SOAPMessage msg) {
    SOAPMessage message = null;
    try {
        message = fac.createMessage();

        SOAPPart part = message.getSOAPPart();
        SOAPEnvelope envelope = part.getEnvelope();
        SOAPBody body = envelope.getBody();
```

```
Name bodyName = envelope.createName("price-list",
    "PriceList", "http://sonata.coffeebreak.com");
SOAPBodyElement list = body.addBodyElement(bodyName);

Name coffeeN = envelope.createName("coffee");
SOAPElement coffee = list.addChildElement(coffeeN);

Name coffeeNm1 = envelope.createName("coffee-name");
SOAPElement coffeeName =
    coffee.addChildElement(coffeeNm1);
coffeeName.addTextNode("Arabica");

Name priceName1 = envelope.createName("price");
SOAPElement price1 = coffee.addChildElement(priceName1);
price1.addTextNode("4.50");

Name coffeeNm2 = envelope.createName("coffee-name");
SOAPElement coffeeName2 =
    coffee.addChildElement(coffeeNm2);
coffeeName2.addTextNode("Espresso");

Name priceName2 = envelope.createName("price");
SOAPElement price2 = coffee.addChildElement(priceName2);
price2.addTextNode("5.00");

Name coffeeNm3 = envelope.createName("coffee-name");
SOAPElement coffeeName3 =
    coffee.addChildElement(coffeeNm3);
coffeeName3.addTextNode("Dorada");

Name priceName3 = envelope.createName("price");
SOAPElement price3 = coffee.addChildElement(priceName3);
price3.addTextNode("6.00");

Name coffeeNm4 = envelope.createName("coffee-name");
SOAPElement coffeeName4 =
    coffee.addChildElement(coffeeNm4);
coffeeName4.addTextNode("House Blend");

Name priceName4 = envelope.createName("price");
SOAPElement price4 = coffee.addChildElement(priceName4);
price4.addTextNode("5.00");

message.saveChanges();

} catch(Exception e) {
    e.printStackTrace();
}
```

```

    }
    return message;
  }
}

```

Returning the Order Confirmation

ConfirmationServlet creates the confirmation message that is returned to the call method that is invoked in OrderRequest. It is very similar to the code in PriceListServlet except that instead of building a price list, its onMessage method builds a confirmation with the order number and shipping date.

The onMessage method for this servlet uses the SOAPMessage object passed to it by the doPost method to get the order number sent in OrderRequest. Then it builds a confirmation message with the order ID and shipping date. The shipping date is calculated as today's date plus two days.

```

public SOAPMessage onMessage(SOAPMessage message) {

    SOAPMessage confirmation = null;

    try {

        //retrieve the orderID element from the message received
        SOAPBody sentSB = message.getSOAPPart().
            getEnvelope().getBody();
        Iterator sentIt = sentSB.getChildElements();
        SOAPBodyElement sentSBE =
            (SOAPBodyElement)sentIt.next();
        Iterator sentIt2 = sentSBE.getChildElements();
        SOAPElement sentSE = (SOAPElement)sentIt2.next();

        //get the text for orderID to put in confirmation
        String sentID = sentSE.getValue();

        //create the confirmation message
        confirmation = fac.createMessage();
        SOAPPart sp = confirmation.getSOAPPart();
        SOAPEnvelope env = sp.getEnvelope();
        SOAPBody sb = env.getBody();
        Name newBodyName = env.createName("confirmation",
            "Confirm", "http://sonata.coffeebreak.com");
        SOAPBodyElement confirm =
            sb.addBodyElement(newBodyName);

        //create the orderID element for confirmation
    }
}

```

```

Name newOrderIDName = env.createName("orderId");
SOAPElement newOrderNo =
    confirm.addChildElement(newOrderIDName);
newOrderNo.addTextNode(sentID);

//create ship-date element
Name shipDateName = env.createName("ship-date");
SOAPElement shipDate =
    confirm.addChildElement(shipDateName);

//create the shipping date
Date today = new Date();
long msPerDay = 1000 * 60 * 60 * 24;
long msTarget = today.getTime();
long msSum = msTarget + (msPerDay * 2);
Date result = new Date();
result.setTime(msSum);
String sd = result.toString();
shipDate.addTextNode(sd);

confirmation.saveChanges();

} catch (Exception ex) {
    ex.printStackTrace();
}
return confirmation;
}

```

Coffee Break Server

The Coffee Break Server uses servlets, JSP pages, and JavaBeans components to dynamically construct HTML pages for consumption by a Web browser client. The JSP pages use the template tag library discussed in A Template Tag Library (page 141) to achieve a common look and feel among the HTML pages, and many of the JSTL custom tags discussed in Chapter 6 to minimize the use of scripting.

The Coffee Break Server implementation is organized along the Model-View-Controller design pattern. The Dispatcher servlet is the controller. It examines the request URL, creates and initializes model JavaBeans components, and dispatches requests to view JSP pages. The JavaBeans components contain the business logic for the application—they call the Web services and perform computations on the data returned from the services. The JSP pages format the data

stored in the JavaBeans components. The mapping between JavaBeans components and pages is summarized in Table 12–1.

Table 12–1 Model and View Components

Function	JSP Page	JavaBeans Component
Update order data	orderForm	ShoppingCart
Update delivery and billing data	checkoutForm	CheckoutFormBean
Display order confirmation	checkoutAck	OrderConfirmations

To browse the code for the Coffee Break server in the IDE, mount the filesystem `<INSTALL>/j2eetutorial/examples/cb/cbservice`.

Service-Oriented JavaBeans Components

The Coffee Break server uses the following JavaBeans components to represent data returned from the JAX-RPC and JAXM Web services:

- AddressBean - shipping information for customer
- ConfirmationBean - order id and ship date
- CustomerBean - customer contact information
- LineItemBean - order item
- OrderBean - order id, customer, address, list of line items, total price
- PriceItemBean - price list entry (coffee name and wholesale price)
- PriceListBean - price list

The components are contained in the `cbservice` package, which is found in the directory `<INSTALL>/j2eetutorial/examples/cb/WEB-INF/classes/cbservice`

JAX-RPC Client

The JAX-RPC client is generated directly from the Web service and is located in `<INSTALL>/j2eetutorial/examples/cb/jaxrpc/clientutil`. Since the JAX-RPC client returns JavaBeans components defined in the `clientutil` package under the `jaxrpc` directory and the Coffee Break server class that accesses those components (see `CheckoutFormBean`, page 412) uses types defined in the `cbSERVICE` package, `CheckoutFormBean` converts `clientutil` types to equivalent types defined in the `cbSERVICE` package.

JAXM Client

The Coffee Break server sends requests to its JAXM distributor. Because the request-response form of JAXM messaging is being used, the client applications use the `SOAPConnection` method `call` to send messages.

```
SOAPMessage response = con.call(request, endpoint);
```

Accordingly, the client code has two major tasks. The first is to create and send the request; the second is to extract the content from the response. These tasks are handled by the classes `<INSTALL>/j2eetutorial/examples/cb/WEB-INF/classes/cbSERVICE/JAXMPriceListRequest` and `<INSTALL>/j2eetutorial/examples/cb/WEB-INF/classes/cbSERVICE/JAXMOrderRequest`.

Sending the Request

This section covers the code for creating and sending the request for an updated price list. This is done in the `getPriceList` method of `JAXMPriceListRequest`, which follows the DTD `price-list.dtd`.

The `getPriceList` method begins by creating the connection that will be used to send the request. Then it gets the default `MessageFactory` object so that it can create the `SOAPMessage` object `msg`.

```
SOAPConnectionFactory scf =
    SOAPConnectionFactory.newInstance();
SOAPConnection con = scf.createConnection();

MessageFactory mf = MessageFactory.newInstance();
SOAPMessage msg = mf.createMessage();
```

The next step is to access the message's `SOAPEnvelope` object, which will be used to create a `Name` object for each new element that is created. It is also used to access the `SOAPBody` object, to which the message's content will be added.

```
SOAPPart part = msg.getSOAPPart();
SOAPEnvelope envelope = part.getEnvelope();
SOAPBody body = envelope.getBody();
```

The file `price-list.dtd` specifies that the top-most element inside the body is `request-prices` and that it contains the element `request`. The text node added to `request` is the text of the request being sent. Every new element that is added to the message must have a `Name` object to identify it, which is created by the `Envelope` method `createName`. The following lines of code create the top-level element in the `SOAPBody` object `body`. The first element created in a `SOAPBody` object is always a `SOAPBodyElement` object.

```
Name bodyName = envelope.createName("request-prices",
    "RequestPrices", "http://sonata.coffeebreak.com");
SOAPBodyElement requestPrices =
    body.addBodyElement(bodyName);
```

In the next few lines, the code adds the element `request` to the element `request-prices` (represented by the `SOAPBodyElement` `requestPrices`.) Then the code adds a text node containing the text of the request. Next, because there are no other elements in the request, the code calls the method `saveChanges` on the message to save what has been done.

```
Name requestName = envelope.createName("request");
SOAPElement request =
    requestPrices.addChildElement(requestName);
request.addTextNode("Send updated price list.");

msg.saveChanges();
```

With the creation of the request message completed, the code sends the message to the JAXM coffee supplier. The message being sent is the `SOAPMessage` object `msg`, to which the elements created in the previous code snippets were added. The endpoint is the URI for the JAXM coffee supplier. The `SOAPConnection` object `con` is used to send the message, and because it is no longer needed, it is closed.

```
URL endpoint = new URL(url);
SOAPMessage response = con.call(msg, endpoint);
con.close();
```


When the `call` method is executed, the application server executes the servlet `PriceListServlet`. This servlet creates and returns a `SOAPMessage` object whose content is the JAXM distributor's price list. (`PriceListServlet` is discussed in *Returning the Price List*, page 395.) The application server knows to execute `PriceListServlet` because the `web.xml` file at `<INSTALL>/j2eetutorial/examples/cb/jaxm/service/WEB-INF` maps the given endpoint to that servlet.

Extracting the Price List

This section demonstrates (1) retrieving the price list that is contained in response, the `SOAPMessage` object returned by the method call, and (2) returning the price list as a `PriceListBean`.

The code creates an empty `Vector` object that will hold the coffee-name and price elements that are extracted from response. Then the code uses response to access its `SOAPBody` object, which holds the message's content. Notice that the `SOAPEnvelope` object is not accessed separately because it is not needed for creating Name objects, as it was in the previous section.

```
Vector list = new Vector();
SOAPBody responseBody = response.getSOAPPart().
    getEnvelope().getBody();
```

The next step is to retrieve the `SOAPBodyElement` object. The method `getChildElements` returns an `Iterator` object that contains all of the child elements of the element on which it is called, so in the following lines of code, `it1` contains the `SOAPBodyElement` object `bodyE1`, which represents the price-list element.

```
Iterator it1 = responseBody.getChildElements();
while (it1.hasNext()) {
    SOAPBodyElement bodyE1 = (SOAPBodyElement)it1.next();
```

The `Iterator` object `it2` holds the child elements of `bodyE1`, which represent coffee elements. Calling the method `next` on `it2` retrieves the first coffee element in `bodyE1`. As long as `it2` has another element, the method `next` will return the next coffee element.

```
Iterator it2 = bodyE1.getChildElements();
while (it2.hasNext()) {
    SOAPElement child2 = (SOAPElement)it2.next();
```

The next lines of code drill down another level to retrieve the coffee-name and price elements contained in `it3`. Then the message `getValue` retrieves the text (a coffee name or a price) that the JAXM coffee distributor added to the coffee-name and price elements when it gave content to response. The final line in the following code fragment adds the coffee name or price to the Vector object `list`. Note that because of the nested while loops, for each coffee element that the code retrieves, both of its child elements (the coffee-name and price elements) are retrieved.

```

        Iterator it3 = child2.getChildElements();
        while (it3.hasNext()) {
            SOAPElement child3 = (SOAPElement)it3.next();
            String value = child3.getValue();
            list.addElement(value);
        }
    }
}

```

The last code fragment adds the coffee names and their prices (as a `PriceListItem`) to the `ArrayList` `priceItems`, and prints each pair on a separate line. Finally it constructs and returns a `PriceListBean`.

```

    ArrayList priceItems = new ArrayList();

    for (int i = 0; i < list.size(); i = i + 2) {
        new PriceItemBean();
        pib.setCoffeeName(list.elementAt(i).toString());
        pib.setPricePerPound(new
            BigDecimal(list.elementAt(i + 1).toString()));
        priceItems.add(pib);
    }
    Date today = new Date();
    Date endDate = DateHelper.addDays(today, 30);
    plb = new PriceListBean();
    plb.setStartDate(today);
    plb.setEndDate(endDate);
    plb.setPriceItems(priceItems);
}

```

Ordering Coffee

The other kind of message that the Coffee Break server can send to the JAXM distributor is an order for coffee. This is done in the `placeOrder` method of `JAXMOrderRequest`, which follows the DTD `coffee-order.dtd`.

Creating the Order

As with the client code for requesting a price list, the `placeOrder` method starts out by creating a `SOAPConnection` object, creating a `SOAPMessage` object, and accessing the message's `SOAPEnvelope` and `SOAPBody` objects.

```
SOAPConnectionFactory scf =
    SOAPConnectionFactory.newInstance();
SOAPConnection con = scf.createConnection();
MessageFactory mf = MessageFactory.newInstance();
SOAPMessage msg = mf.createMessage();

SOAPPart part = msg.getSOAPPart();
SOAPEnvelope envelope = part.getEnvelope();
SOAPBody body = envelope.getBody();
```

Next the code creates and adds XML elements to form the order. As is required, the first element is a `SOAPBodyElement`, which in this case is `coffee-order`.

```
Name bodyName = envelope.createName("coffee-order", "PO",
    "http://sonata.coffeebreak.com");
SOAPBodyElement order = body.addBodyElement(bodyName);
```

The application then adds the next level of elements, the first of these being `orderId`. The value given to `orderId` is extracted from the `OrderBean` object passed to the `OrderRequest.placeOrder` method.

```
Name orderIdName = envelope.createName("orderId");
SOAPElement orderId = order.addChildElement(orderIDName);
orderId.addTextNode(orderBean.getId());
```

The next element, `customer`, has several child elements that give information about the customer. This information is also extracted from the `Customer` component of `OrderBean`.

```
Name childName = envelope.createName("customer");
SOAPElement customer = order.addChildElement(childName);

childName = envelope.createName("last-name");
SOAPElement lastName = customer.addChildElement(childName);
lastName.addTextNode(orderBean.getCustomer().
    getLastName());

childName = envelope.createName("first-name");
SOAPElement firstName = customer.addChildElement(childName);
firstName.addTextNode(orderBean.getCustomer().
```

```

    getFirstName());

    childName = envelope.createName("phone-number");
    SOAPElement phoneNumber = customer.addChildElement(childName);
    phoneNumber.addTextNode(orderBean.getCustomer().
        getPhoneNumber());

    childName = envelope.createName("email-address");
    SOAPElement emailAddress =
        customer.addChildElement(childName);
    emailAddress.addTextNode(orderBean.getCustomer().
        getEmailAddress());

```

The address element, added next, has child elements for the street, city, state, and zip code. This information is extracted from the Address component of OrderBean.

```

    childName = envelope.createName("address");
    SOAPElement address = order.addChildElement(childName);

    childName = envelope.createName("street");
    SOAPElement street = address.addChildElement(childName);
    street.addTextNode(orderBean.getAddress().getStreet());

    childName = envelope.createName("city");
    SOAPElement city = address.addChildElement(childName);
    city.addTextNode(orderBean.getAddress().getCity());

    childName = envelope.createName("state");
    SOAPElement state = address.addChildElement(childName);
    state.addTextNode(orderBean.getAddress().getState());

    childName = envelope.createName("zip");
    SOAPElement zip = address.addChildElement(childName);
    zip.addTextNode(orderBean.getAddress().getZip());

```

The element line-item has three child elements: coffeeName, pounds, and price. This information is extracted from the LineItems list contained in OrderBean.

```

    for (Iterator it = orderBean.getLineItems().iterator();
        it.hasNext(); ; ) {
        LineItemBean lib = (LineItemBean)it.next();

        childName = envelope.createName("line-item");
        SOAPElement lineItem =
            order.addChildElement(childName);
    }

```

```

    childName = envelope.createName("coffeeName");
    SOAPElement coffeeName =
        lineItem.addChildElement(childName);
    coffeeName.addTextNode(lib.getCoffeeName());

    childName = envelope.createName("pounds");
    SOAPElement pounds =
        lineItem.addChildElement(childName);
    pounds.addTextNode(lib.getPounds().toString());

    childName = envelope.createName("price");
    SOAPElement price =
        lineItem.addChildElement(childName);
    price.addTextNode(lib.getPrice().toString());
}
//total
childName = envelope.createName("total");
SOAPElement total =
    order.addChildElement(childName);
total.addTextNode(orderBean.getTotal().toString());
}

```

With the order complete, the application sends the message and closes the connection.

```

URL endpoint = new URL(url);
SOAPMessage reply = con.call(msg, endpoint);
con.close();

```

Because the `web.xml` file maps the given endpoint to `ConfirmationServlet`, the application server executes that servlet (discussed in [Returning the Order Confirmation](#), page 400) to create and return the `SOAPMessage` object `reply`.

Retrieving the Order Confirmation

The rest of the `placeOrder` method retrieves the information returned in `reply`. The client knows what elements are in it because they are specified in `confirm.dtd`. After accessing the `SOAPBody` object, the code retrieves the confirmation element and gets the text of the `orderId` and `ship-date` ele-

ments. Finally, it constructs and returns a `ConfirmationBean` with this information.

```
SOAPBody sBody = reply.getSOAPPart().getEnvelope().getBody();
Iterator bodyIt = sBody.getChildElements();
SOAPBodyElement sbEl = (SOAPBodyElement)bodyIt.next();
Iterator bodyIt2 = sbEl.getChildElements();

SOAPElement ID = (SOAPElement)bodyIt2.next();
String id = ID.getValue();

SOAPElement sDate = (SOAPElement)bodyIt2.next();
String shippingDate = sDate.getValue();
Date date = df.parse(shippingDate);
Calendar cal = new GregorianCalendar();
cal.setTime(date);
cb = new ConfirmationBean();
cb.setOrderId(id);
cb.setShippingDate(cal);
```

JSP Pages

orderForm

`orderForm` displays the current contents of the shopping cart. The first time the page is requested, the quantities of all the coffees are 0. Each time the customer changes the coffees amounts and clicks the Update button, the request is posted back to `orderForm`. The `Dispatcher` servlet updates the values in the shopping cart, which are then redisplayed by `orderForm`. When the order is complete, the customer proceeds to the `checkoutForm` page by clicking the Checkout link.

checkoutForm

`checkoutForm` is used to collect delivery and billing information for the customer. When the Submit button is clicked, the request is posted to the `checkoutAck` page. However, the request is first handled by the `Dispatcher`, which invokes the `validate` method of `checkoutFormBean`. If the validation does not succeed, the requested page is reset to `checkoutForm`, with error notifications in each invalid field. If the validation succeeds, `checkoutFormBean` submits suborders to each distributor and stores the result in the request-scoped `OrderConfirmations` JavaBeans component and control is passed to `checkoutAck`.

checkoutAck

checkoutAck simply displays the contents of the OrderConfirmations JavaBeans component, which is a list of the suborders comprising an order and the ship dates of each suborder.

JavaBeans Components

RetailPriceList

RetailPriceList is a list of retail price items. A retail price item contains a coffee name, a wholesale price per pound, a retail price per pound, and a distributor. This data is used for two purposes: it contains the price list presented to the end user and is used by CheckoutFormBean when it constructs the suborders dispatched to coffee distributors.

It first performs a JAXR lookup to determine the JAX-RPC service endpoints. It then queries each JAX-RPC service for a coffee price list. Finally it queries the JAXM service for a price list. The two price lists are combined and a retail price per pound is determined by adding a 35% markup to the wholesale prices.

Discovering the JAX-RPC Service

Instantiated by RetailPriceList, JAXRQueryByName connects to the registry server and searches for coffee distributors registered with the name JAXRPCCoffeeDistributor in the executeQuery method. The method returns a collection of organizations which contain services. Each service is accessible via a service binding or URI. RetailPriceList makes a JAX-RPC call to each URI.

ShoppingCartItem

ShoppingCart is a list of shopping cart items. A shopping cart item contains a retail price item, the number of pounds of that item, and the total price for that item.

OrderConfirmation

OrderConfirmations is a list of order confirmation objects. An order confirmation contains order and confirmation objects.

CheckoutFormBean

CheckoutFormBean checks the completeness of information entered into checkoutForm. If the information is incomplete, the bean populates error messages and Dispatcher redisplay checkoutForm with the error messages. If the information is complete, order requests are constructed from the shopping cart and the information supplied to checkoutForm and are sent to each distributor. As each confirmation is received, an order confirmation is created and added to OrderConfirmations. Note that since the JAX-RPC client returns JavaBeans components defined in the clientutil package, CheckoutFormBean converts types returned from the JAX-RPC client to equivalent types defined in the cbservice package.

```

if (allOk) {
    String orderId = CCNumber;
    cbconfirmation = new ConfirmationBean();
    cbconfirmation.setOrderId(orderId);
    AddressBean cbaddress = new AddressBean();
    cbaddress.setStreet(street);
    cbaddress.setCity(city);
    cbaddress.setState(state);
    cbaddress.setZip(zip);
    CustomerBean cbcustomer= new CustomerBean();
    cbcustomer.setFirstName(firstName);
    cbcustomer.setLastName(lastName);
    cbcustomer.setPhoneNumber("(" + areaCode+ ") " +
        + phoneNumber);
    cbcustomer.setEmailAddress(email);

    for(Iterator d = rpl.getDistributors().iterator();
        d.hasNext(); ) {
        String distributor = (String)d.next();
        System.out.println(distributor);
        ArrayList lis = new ArrayList();
        BigDecimal price = new BigDecimal("0.00");
        BigDecimal total = new BigDecimal("0.00");
        for(Iterator c = cart.getItems().iterator();
            c.hasNext(); ) {
            ShoppingCartItem sci = (ShoppingCartItem) c.next();
            if ((sci.getItem().getDistributor()).
                equals(distributor) &&
                sci.getPounds().floatValue() > 0) {
                price = sci.getItem().
                    getWholesalePricePerPound().
                    multiply(sci.getPounds());
                total = total.add(price);
            }
        }
    }
}

```



```

        LineItemBean li = new LineItemBean();
        li.setCoffeeName(sci.getItem().getCoffeeName());
        li.setPounds(sci.getPounds(),);
        li.setPrice(sci.getItem().
getWholesalePricePerPound());
        lis.add(li);
    }
}

if (!lis.isEmpty()) {
    OrderBean cborder = new OrderBean();
    cborder.setId(orderId);
    cborder.setCustomer(cbcustomer);
    cborder.setLineItems(lis);
    cborder.setTotal(total);
    cborder.setAddress(cbaddress);
    String cbpropsName = "cbservice.CoffeeServices";

    ResourceBundle cbpropBundle =
        ResourceBundle.getBundle(cbpropsName);

    String JAXMOrderURL = cbpropBundle.
        getString("JAXMOrder.url");

    if (distributor.equals(JAXMOrderURL)) {
        JAXMOrderRequest or =
            new JAXMOrderRequest(JAXMOrderURL);
        cbconfirmation = or.placeOrder(cborder);
    } else {
        clientutil.OrderGenClient.
            CustomerBean customer =
                new clientutil.OrderGenClient.
                    CustomerBean();
        customer.setFirstName(firstName);
        customer.setLastName(lastName);
        customer.setPhoneNumber("(" + areaCode+ ") " +
            phoneNumber);
        customer.setEmailAddress(email);
        clientutil.OrderGenClient.
            AddressBean address =
                new clientutil.OrderGenClient.
                    AddressBean();
        address.setStreet(street);
        address.setCity(city);
        address.setState(state);
        address.setZip(zip);
        clientutil.OrderGenClient.
            OrderBean order =

```

```

        new clientutil.OrderGenClient.
        OrderBean();
    ArrayList newlis = new ArrayList();
    for(Iterator c = newlis.iterator();c.hasNext();) {
        LineItemBean li = (LineItemBean) c.next();
        clientutil.OrderGenClient.
        LineItemBean lib =
            new clientutil.OrderGenClient.
            LineItemBean();
        lib.setCoffeeName(li.getCoffeeName());
        lib.setPounds(li.getPounds());
        lib.setPrice(li.getPrice());
        newlis.add(lib);
    }
    order.setId(orderId);
    order.setCustomer(customer);
    order.setLineItems(newlis);
    order.setTotal(total);
    order.setAddress(address);
    OrderCaller ocaller =
        new OrderCaller(distributor);
    clientutil.OrderGenClient.
    ConfirmationBean confirmation =
        = ocaller.placeOrder(order);
    cbconfirmation.setShippingDate(
        confirmation.getShippingDate());
    }
    OrderConfirmation oc = new
        OrderConfirmation(cborder, cbconfirmation);
    ocs.add(oc);
    }
}
}

```

RetailPriceListServlet

The `RetailPriceListServlet` responds to requests to reload the price list via the URL `/loadPriceList`. It simply creates a new `RetailPriceList` and a new `ShoppingCart`.

Since this servlet would be used by administrators of the Coffee Break Server, it is a protected Web resource. In order to load the price list, a user must authenticate (using basic authentication) and the authenticated user must be in the `admin` role.

To view the security settings for `cb` service:

1. In the IDE, mount the filesystem
`<INSTALL>/j2eetutorial/examples/cb/cbservice`.
2. Expand the `WEB-INF` directory.
3. Select the `web.xml` file.
4. Select the Security property sheet. You will notice that the login configuration is specified to be BASIC, a security role `admin` is defined, and a security constraint allows the `admin` role to access the resources in the `WRCollection`. If you edit `WRCollection`, you will notice that it contains the URL `/loadPriceList`.

Deploying and Running the Application

The source code for the Coffee Break application is located in the directory `<INSTALL>/j2eetutorial/examples/cb`. Within the `cb` directory are subdirectories for each Web application—`jaxm`, `jaxrpc`, `cb` service.

Add Web services JARs to the compilation classpath. To do so:

1. Choose Options from the Tools menu.
2. Expand the Building node, then the Compiler Types node.
3. Choose External Compilation.
4. Select the Expert tab.
5. Click the Class Path property and open the property editor.
6. Click Add JAR/Zip.
7. In the file chooser, navigate to the `<S1AS7_HOME>/share/lib` directory and choose the `jaxr-api.jar` and `saaj-api.jar` files.
8. Click OK.
9. Click OK in the property editor, then click Close in the Options window.

The JARs are also located in the `<S1STUDIO_HOME>/jwsdp/common/lib` directory, so you may specify that location for the classpath if you prefer.

Building, Deploying, and Registering the JAX-RPC Service

This section contains step-by-step instructions for the following tasks:

- Creating the JAX-RPC Service
- Specifying the Serializers
- Generating the Service's WSDL File, Serializers, and other Helper Classes
- Deploying the JAX-RPC Service
- Testing the JAX-RPC Service
- Registering the JAX-RPC Service

Creating the JAX-RPC Service

1. In the IDE, mount this filesystem:
`<INSTALL>/j2eetutorial/examples/cb/jaxrpc`
2. Expand the `jaxrpc` node.
3. Compile the beans, registry, and service packages.

The beans package contains JavaBeans components that represent business information, such as `AddressBean`. In the JAX-RPC service's methods, these beans are parameters or return types.

The registry package holds the JAXR programs that publish the service in (and remove it from) the registry.

The service package contains `DateHelper`, a utility class; `PriceLoader`, which loads coffee prices into a list; and `Supplier`, which implements the service's remote procedures.

4. Right-click the service package and choose `New→Web Services→Web Service`.
5. In the wizard's Specify Web Service pane, do the following.
 - a. In the Name field, enter `SupplierService`.
 - b. In the Package field, enter `service`.
 - c. For the Create From buttons, choose `Java Methods`.
 - d. For the Architecture buttons, choose `Web centric`.
 - e. Click `Next`.

6. In the wizard's Select Methods pane, do the following.
 - a. Expand the nodes in the service package until you see the methods beneath the `Supplier` class.
 - b. Choose the `getPriceList` and `placeOrder` methods. These are the methods that remote clients may call.
 - c. Click Finish.

If the "Collection of What?" window appears, click Cancel. In Sun ONE Studio 4 update 1, a bug prevents this operation from working correctly.

In the Explorer, the IDE displays a Web service node (a cube containing a sphere) for the `SupplierService`.

Specifying the Serializers

The JAX-RPC runtime system uses serializers when converting Java objects into XML data types. The runtime system needs a serializer for every parameter and return type of the service's methods. The IDE generates the serializers automatically when you perform the Generate Web Service task, as in the next section. The IDE identifies the serializers it needs to generate by examining the classes of the parameters and return types. However, if such a class contains a collection, the IDE cannot automatically identify the type of the collection's members. In this case, you must specify the collection's member type so that the IDE can generate a serializer for the member.

In this example, the `getPriceList` method returns a `PriceListBean`, a collection made up of `PriceItemBean` components. The `placeOrder` method passes an `OrderBean` parameter, which contains a collection of `LineItemBean` components. To specify serializers for `LineItemBean` and `PriceItemBean`, perform these steps:

1. Right-click the Web service node for `SupplierService` (a cube containing a sphere) and choose Properties.
2. In the properties sheet, choose Serialization Classes and open the properties editor.
3. In the properties editor, add the `beans.LineItemBean` and `beans.PriceItemBean` classes.
4. Click OK to close the properties editor.
5. Close the properties sheet.

Generating the Service's WSDL File, Serializers, and other Helper Classes

To generate these files and classes, right-click the Web service node for `SupplierService` and choose `Generate Web Service`. The IDE creates the following:

- The `SupplierService` WSDL file, represented by a rectangle with a sphere in the lower-left corner
- Serializers for the JavaBeans components, placing these serializers in beans package
- The `SupplierServiceGenServer` package, which contains the other serializers and the helper classes

Deploying the JAX-RPC Service

Right-click the Web service node for `SupplierService` (a cube containing a sphere) and choose `Deploy`.

Testing the JAX-RPC Service

To test the service, you create a Web service client and run the `TestOrderCaller` and `TestPriceFetcher` programs. Contained in the `test` package, these programs are static stub clients. (For more information, see the section `Static Stub Client Example`, page 277.)

Note: The test programs assume that you have deployed `SupplierService` on `localhost`. If you are running the service on a different host, you need to update the service URLs.

1. Right-click the `clientutil` package and choose `File→New→Web Services→Web Service Client`.
2. In the wizard's `Specify Web Service Client` pane, do the following:
 - a. In the `Name` field enter `Order`.
 - b. In the `Package` field, enter `clientutil`.
 - c. For the `Create From` buttons, choose `Local WSDL File`.
 - d. Click `Next`.

3. In the wizard's Select Local WSDL File pane, expand the service package and choose the SupplierService WSDL node (a rectangle with a sphere in the lower-left corner)
4. Click Finish.
5. Right-click the Order client node and choose Generate Client Proxy.

This action creates the OrderGenClient package, which contains the stub class, serializer classes, and other helper classes required by the client at runtime.

6. In the test package, execute TestOrderCaller. The output window should display:


```
orderId = 123
shippingDate = mm/dd/yy (tomorrow's date)
```
7. . Execute TestPriceFetcher. The output window should display:

```
mm/dd/yy (today's date) mm/dd/yy (60 days from today)
Mocca 4.00
Wake Up Call 5.50
French Roast 5.00
Kona 6.50
```

Registering the JAX-RPC Service

1. Start the UDDI Registry.
 - a. Select the Runtime tab of the Explorer.
 - b. Expand the UDDI Server Registry node.
 - c. Right-click the Internal UDDI Registry node and choose Start Server.
2. Register the service with the Registry Server.
 - a. Select the Filesystems tab of the Explorer.
 - b. In the jaxrpc subdirectory, expand the registry package.
 - c. Execute the OrgPublisher program.
 - d. The registration process can take some time, so wait until you see the following output before proceeding:

```
Created connection to registry
Got registry service, query manager, and life cycle manager
Established security credentials
Organization saved
Organization key is xxxxxxxx
```

Later on, you can remove the JAX-RPC service from the registry by executing `OrgRemover`.

Deploying the JAXM Service

To build and deploy the JAXM service:

1. In the IDE, mount the filesystem
`<INSTALL>/j2eetutorial/examples/cb/jaxm/service`.
2. Right-click the WEB-INF directory and choose Deploy.

Testing the JAXM Service

To test the JAXM service, you run the test programs `TestPriceListRequest` or `TestOrderRequest` in the `<INSTALL>/j2eetutorial/examples/cb/jaxm/test` directory.

Note: The test programs assume that you have deployed the JAXM service on `localhost`. If you are running the service on a different host, you need to update the service URLs.

To run the test programs in the IDE:

1. In the IDE, mount the filesystem
`<INSTALL>/j2eetutorial/examples/cb/jaxm`.
2. Right-click `TestPriceListRequest` or `TestOrderRequest` and choose Execute. Here is what you should see when you run the former:

Arabica	4.50
Espresso	5.00
Dorada	6.00
House Blend	5.00

Deploying the Coffee Break Server

Server Configuration

The Coffee Break Server accesses three services—registry, JAX-RPC service, JAXM service—to build retail price lists and order coffee. The information needed to access the services is stored in the file `<INSTALL>/examples/cb/cbservice/CoffeeServices.properties`. This file contains URLs and security parameters for accessing the registry, the name under which the JAX-RPC service is registered, and the URLs for making requests on the JAXM service.

```
registryQuery.url=http://localhost:8089/registry-  
server/RegistryServerServlet  
registryPublish.url=http://localhost:8089/registry-  
server/RegistryServerServlet  
registry.username=testuser  
registry.password=testuser  
JAXRPCService.name=JAXRPCCoffeeDistributor  
JAXMPricelist.url=http://localhost:80/jaxm-coffee-  
supplier/getPriceList  
JAXMOrder.url=http://localhost:80/jaxm-coffee-  
supplier/orderCoffee
```

Note: The default configuration assumes that you are running the registry server and JAXM service on localhost. If you are running either service on a different host, you need to update the service URLs.

Importing the JAX-RPC Client

The Coffee Break server imports the JAX-RPC client classes in the package `clientutil`. To make the classes available for compilation and deployment:

1. Make sure the filesystem `<INSTALL>/j2eetutorial/examples/cb/jaxrpc` is mounted. This is required so that the IDE includes the `clientutil` subdirectory of `jaxrpc` in the classpath during compilation.
2. In the IDE, mount the filesystem `<INSTALL>/j2eetutorial/examples/cb/cbservice`.
3. Expand the `cbservice` node.

4. Add the `clientutil` subdirectory of `jaxrpc` to the Web module as an extra file. When the IDE packages the `cbService` Web module it will include the `clientutil` directory. To add the directory as an extra file:
 - a. Right-click the `WEB-INF` directory.
 - b. Select the Archive property sheet.
 - c. Click the Extra Files property and open the property editor.
 - d. In the Chosen Content pane, select the `WEB-INF/classes` Directory Prefix from the drop-down list.
 - e. In the Source pane, expand the `<INSTALL>/j2eetutorial/examples/cb/jaxrpc` node.
 - f. Select the `clientutil` node.
 - g. Click Add.
 - h. Click OK.
5. Right-click the `WEB-INF` node and choose View WAR Content to check that the `clientutil` package was added.

Deploying the Coffee Break Server.

To deploy the Coffee Break server, right-click the `WEB-INF` directory of `<INSTALL>/j2eetutorial/examples/cb/cbService` and choose Deploy.

Running the Coffee Break Client

After you have deployed all the Web applications, check that all the applications—`SupplierService`, `jaxm-coffee-supplier`, `cbService`—are running by viewing the deployed Web applications with the IDE or the application server administration tool.

You may need to add `<SIAS7_HOME>/share/lib/xercesImpl.jar` to the application server classpath suffix and restart the application server using the application server administration tool. To do so:

1. Open the URL `http://localhost:4848` in a browser.
2. Select the `server1` node.
3. Select the JVM Settings tab.
4. Click the Path Settings link.

5. Add `<${AS7_HOME}>/share/lib/xercesImpl.jar` to the Classpath Suffix text area.
6. Click Save.
7. Click the General tab.
8. Stop and restart the server.

To run the Coffee Break client, open the Coffee Break server URL in a Web browser:

`http://localhost:80/cbservice/orderForm`

You should see a page something like the one shown in Figure 12–2.

Coffee Break

Enter the amount of coffee and click Update to update the totals.
Click Checkout to proceed with your order.

OrderForm			
Coffee	Price	Quantity	Total
Wake Up Call	\$7.43	<input type="text" value="0.0"/>	\$0.00
Mocca	\$5.40	<input type="text" value="0.0"/>	\$0.00
Kona	\$8.78	<input type="text" value="0.0"/>	\$0.00
French Roast	\$6.75	<input type="text" value="0.0"/>	\$0.00
Arabica	\$6.08	<input type="text" value="0.0"/>	\$0.00
Espresso	\$6.75	<input type="text" value="0.0"/>	\$0.00
Dorada	\$8.10	<input type="text" value="0.0"/>	\$0.00
House Blend	\$6.75	<input type="text" value="0.0"/>	\$0.00
Checkout <input type="button" value="Update"/>			\$0.00

Copyright © 2002 Sun Microsystems, Inc.

Figure 12–2 Order Form

After you have gone through the application screens, you will get an order confirmation that looks like the one shown in Figure 12–3.

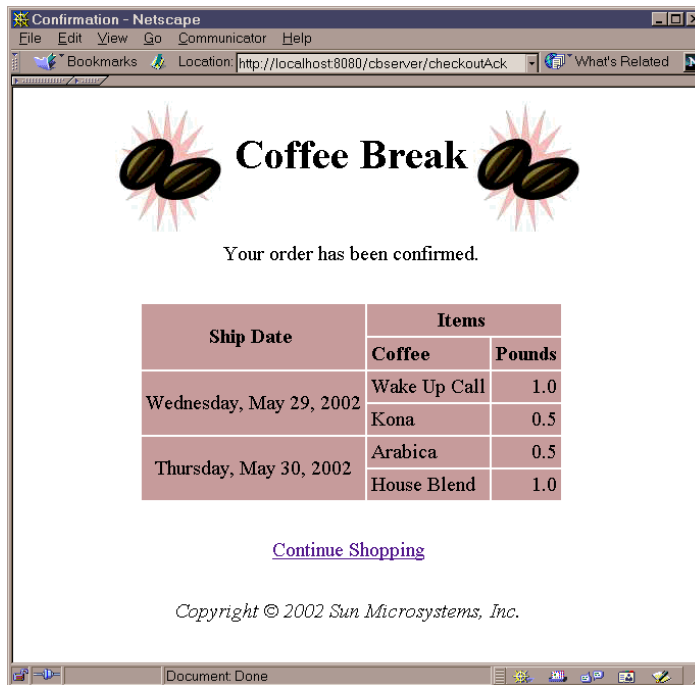


Figure 12–3 Order Confirmation

Java Encoding Schemes

This appendix describes the character-encoding schemes that are supported by the Java platform.

US-ASCII

US-ASCII is a 7-bit encoding scheme that covers the English-language alphabet. It is not large enough to cover the characters used in other languages, however, so it is not very useful for internationalization.

ISO-8859-1

This is the character set for Western European languages. It's an 8-bit encoding scheme in which every encoded character takes exactly 8-bits. (With the remaining character sets, on the other hand, some codes are reserved to signal the start of a multi-byte character.)

UTF-8

UTF-8 is an 8-bit encoding scheme. Characters from the English-language alphabet are all encoded using an 8-bit bytes. Characters for other languages are encoded using 2, 3 or even 4 bytes. UTF-8 therefore produces compact documents for the English language, but for other languages, documents tend to be half again as large as they would be if they used UTF-16. If the majority of a document's text is in a Western European language, then UTF-8 is generally a good choice because it allows for internationalization while still minimizing the space required for encoding.

UTF-16

UTF-16 is a 16-bit encoding scheme. It is large enough to encode all the characters from all the alphabets in the world. It uses 16-bits for most characters, but includes 32-bit characters for ideogram-based languages like Chinese. A Western European-language document that uses UTF-16 will be

twice as large as the same document encoded using UTF-8. But documents written in far Eastern languages will be far smaller using UTF-16.

Note: UTF-16 depends on the system's byte-ordering conventions. Although in most systems, high-order bytes follow low-order bytes in a 16-bit or 32-bit "word", some systems use the reverse order. UTF-16 documents cannot be interchanged between such systems without a conversion.

Further Information

For a complete list of the encodings that can be supported by the Java 2 platform, see:

<http://java.sun.com/j2se/1.3/docs/guide/intl/encoding.doc.html>

B

HTTP Overview

Stephanie Bodoff

MOST Web clients use the HTTP protocol to communicate with a J2EE server. HTTP defines the requests that a client can send to a server and responses that the server can send in reply. Each request contains a URL, which is a string that identifies a Web component or a static object such as an HTML page or image file.

The J2EE server converts an HTTP request to an HTTP request object and delivers it to the Web component identified by the request URL. The Web component fills in an HTTP response object, which the server converts to an HTTP response and sends to the client.

This appendix provides some introductory material on the HTTP protocol. For further information on this protocol, see the Internet RFCs: HTTP/1.0 - RFC 1945, HTTP/1.1 - RFC 2616, which can be downloaded from

<http://www.rfc-editor.org/rfc.html>

HTTP Requests

An HTTP request consists of a request method, a request URL, header fields, and a body. HTTP 1.1 defines the following request methods:

- GET - retrieves the resource identified by the request URL.
- HEAD - returns the headers identified by the request URL.
- POST - sends data of unlimited length to the Web server.
- PUT - stores a resource under the request URL.
- DELETE - removes the resource identified by the request URL.
- OPTIONS - returns the HTTP methods the server supports.
- TRACE - returns the header fields sent with the TRACE request.

HTTP 1.0 includes only the GET, HEAD, and POST methods. Although J2EE servers are only required to support HTTP 1.0, in practice many servers, including the Java WSDP, support HTTP 1.1.

HTTP Responses

An HTTP response contains a result code, header fields, and a body.

The HTTP protocol expects the result code and all header fields to be returned before any body content.

Some commonly used status codes include:

- 404 - indicates that the requested resource is not available.
- 401 - indicates that the request requires HTTP authentication.
- 500 - indicates an error inside the HTTP server which prevented it from fulfilling the request.
- 503 - indicates that the HTTP server is temporarily overloaded, and unable to handle the request.

Glossary

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

access control

The methods by which interactions with resources are limited to collections of users or programs for the purpose of enforcing integrity, confidentiality, or availability constraints.

ACID

The acronym for the four properties guaranteed by transactions: atomicity, consistency, isolation, and durability.

anonymous access

Accessing a resource without authentication.

Ant

A Java-based, and thus cross-platform, build tool that can be extended using Java classes. The configuration files are XML-based, calling out a target tree where various tasks get executed.

applet

A component that typically executes in a Web browser, but can execute in a variety of other applications or devices that support the applet programming model.

Application Deployment Tool

A tool for creating WAR files for application deployment and handling security issues.

archiving

Saving the state of an object and restoring it.

attribute

A qualifier on an XML tag that provides additional information.

authentication

The process that verifies the identity of a user, device, or other entity in a computer system, usually as a prerequisite to allowing access to resources in a system. Java WSDP requires three types of authentication: *basic*, *form-based*, and *mutual*, and supports *digest* authentication.

authorization

The process by which access to a method or resource is determined. Authorization depends upon the determination of whether the principal associated with a request through authentication is in a given security role. A security role is a logical grouping of users defined by the person who assembles the application. A deployer maps security roles to security identities. Security identities may be principals or groups in the operational environment.

authorization constraint

An authorization rule that determines who is permitted to access a Web resource collection.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

B2B

Business-to-business.

basic authentication

An authentication mechanism in which a Web server authenticates an entity with a user name and password obtained using the Web application's built-in authentication mechanism.

binary entity

See *unparsed entity*.

binding

Construction of the code needed to process a well-defined bit of XML data.

build file

The XML file that contains one project that contains one or more targets. A target is a set of tasks you want to be executed. When starting Ant, you can select which target(s) you want to have executed. When no target is given, the project's default is used.

build properties file

A file named `build.properties` that contains properties in

business logic

The code that implements the functionality of an application.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

callback methods

Component methods called by the container to notify the component of important events in its life cycle.

CDATA

A predefined XML tag for Character DATA that means don't interpret these characters, as opposed to Parsed Character Data (PCDATA), in which the normal rules of XML syntax apply (for example, angle brackets demarcate XML tags, tags define XML elements, etc.). CDATA sections are typically used to show examples of XML syntax.

certificate authority

A trusted organization that issues public key certificates and provides identification to the bearer.

client certificate authentication

An authentication mechanism that uses HTTP over SSL, in which the server and, optionally, the client authenticate each other with a public key certificate that conforms to a standard that is defined by X.509 Public Key Infrastructure (PKI).

comment

Text in an XML document that is ignored, unless the parser is specifically told to recognize it.

content

The part of an XML document that occurs after the prolog, including the root element and everything it contains.

commit

The point in a transaction when all updates to any resources involved in the transaction are made permanent.

component

An application-level software unit supported by a *container*. Components are configurable at deployment time. See also *Web components*.

component contract

The contract between a component and its container. The contract includes: life cycle management of the component, a context interface that the instance uses to obtain various information and services from its container, and a list of services that every container must provide for its components.

component-managed sign-on

Security information needed for signing on to the resource to the `getConnection()` method is provided by an application component.

connection

See *resource manager connection*.

connection factory

See *resource manager connection factory*.

connector

A standard extension mechanism for containers to provide connectivity to enterprise information systems. A connector is specific to an enterprise information system and consists of a resource adapter and application development tools for enterprise information system connectivity. The resource adapter is plugged in to a container through its support for system-level contracts defined in the connector architecture.

Connector element

A representation of the interface between external clients sending requests to a particular service.

container

An entity that provides life cycle management, security, deployment, and runtime services to *components*.

container-managed sign-on

Security information needed for signing on to the resource to the `getConnection()` method is supplied by the container.

context attribute

An object bound into the context associated with a servlet.

Context element

A representation of a Web application that is run within a particular virtual host.

context root

A name that gets mapped to the *document root* of a Web application.

credentials

The information describing the security attributes of a *principal*.

CSS

Cascading Style Sheet. A stylesheet used with HTML and XML documents to add a style to all elements marked with a particular tag, for the direction of browsers or other presentation mechanisms.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

data

The contents of an element, generally used when the element does not contain any subelements. When it does, the more general term *content* is generally used. When the only text in an XML structure is contained in simple elements, and elements that have subelements have little or no data mixed in, then that structure is often thought of as XML data, as opposed to an XML document.

document

In general, an XML structure in which one or more elements contains text intermixed with subelements. See also *data*.

DDP

Document-Driven Programming. The use of XML to define applications.

declaration

The very first thing in an XML document, which declares it as XML. The minimal declaration is `<?xml version="1.0"?>`. The declaration is part of the document *prolog*.

declarative security

Mechanisms used in an application that are expressed in a declarative syntax in a deployment descriptor.

delegation

An act whereby one *principal* authorizes another principal to use its identity or privileges with some restrictions.

deployment

The process whereby software is installed into an operational environment.

deployment descriptor

An XML file provided with each module and application that describes how they should be deployed. The deployment descriptor directs a deployment tool to deploy a module or application with specific container options and describes specific configuration requirements that a deployer must resolve.

digest authentication

An authentication mechanism in which a Web application authenticates to a Web server by sending the server a message digest along its HTTP request message. The digest is computed by employing a one-way hash algorithm to a concatenation of the HTTP request message and the client's password. The digest is typically much smaller than the HTTP request, and doesn't contain the password.

distributed application

An application made up of distinct components running in separate runtime environments, usually on different platforms connected via a network. Typical distributed applications are two-tier (client-server), three-tier (client-middleware-server), and multitier (client-multiple middleware-multiple servers).

document root

The top-level directory of a *WAR*. The document root is where JSP pages, client-side classes and archives, and static Web resources are stored.

DOM

Document Object Model. A tree of objects with interfaces for traversing the tree and writing an *XML* version of it.

DTD

Document Type Definition. An optional part of the document prolog, as specified by the *XML* standard. The DTD specifies constraints on the valid tags and tag sequences that can be in the document. The DTD has a number of shortcomings however, which has led to various schema proposals. For example, the DTD entry `<!ELEMENT username (#PCDATA)>` says that the *XML* element called *username* contains *Parsed Character DATA*—that is, text alone, with no other structural elements under it. The DTD includes both the local subset, defined in the current file, and the external subset, which consists of the definitions contained in external *.dtd* files that are referenced in the local subset using a parameter entity.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**ebXML**

Electronic Business XML. A group of specifications designed to enable enterprises to conduct business through the exchange of *XML*-based messages. It is sponsored by OASIS and the United Nations Centre for the Facilitation of Procedures and Practices in Administration, Commerce and Transport (U.N./CEFACT).

element

A unit of *XML* data, delimited by tags. An *XML* element can enclose other elements.

empty tag

A tag that does not enclose any content.

enterprise bean

A component that implements a business task or business entity and resides in an EJB container; either an *entity bean*, *session bean*, or *message-driven bean*.

enterprise information system

The applications that comprise an enterprise's existing system for handling company-wide information. These applications provide an information infrastructure for an enterprise. An enterprise information system offers a well defined set of services to its clients. These services are exposed to clients as local and/or remote interfaces. Examples of enterprise information systems include: enterprise resource planning systems, mainframe transaction processing systems, and legacy database systems.

enterprise information system resource

An entity that provides enterprise information system-specific functionality to its clients. Examples are: a record or set of records in a database system, a business object in an enterprise resource planning system, and a transaction program in a transaction processing system.

entity

A distinct, individual item that can be included in an XML document by referencing it. Such an entity reference can name an entity as small as a character (for example, "<", which references the less-than symbol, or left-angle bracket (<)). An entity reference can also reference an entire document, or external entity, or a collection of DTD definitions (a parameter entity).

entity bean

An enterprise bean that represents persistent data maintained in a database. An entity bean can manage its own persistence or can delegate this function to its container. An entity bean is identified by a primary key. If the container in which an entity bean is hosted crashes, the entity bean, its primary key, and any remote references survive the crash.

entity reference

A reference to an entity that is substituted for the reference when the XML document is parsed. It may reference a predefined entity like < or it may reference one that is defined in the DTD. In the XML data, the reference could be to an entity that is defined in the local subset of the DTD or to an external XML file (an external entity). The DTD can also carve out a segment of DTD specifications and give it a name so that it can be reused (included) at multiple points in the DTD by defining a parameter entity.

error

A SAX parsing error is generally a validation error—in other words, it occurs when an XML document is not valid, although it can also occur if the declaration specifies an XML version that the parser cannot handle. See also: fatal error, warning.

Extensible Markup Language

A markup language that makes data portable.

external entity

An entity that exists as an external XML file, which is included in the XML document using an *entity reference*.

external subset

That part of the DTD that is defined by references to external .dtd files.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**fatal error**

A fatal error occurs in the SAX parser when a document is not well formed, or otherwise cannot be processed. See also: error, warning.

filter

An object that can transform the header and/or content of a request or response. Filters differ from *Web components* in that they usually do not themselves create responses but rather they modify or adapt the requests for a resource, and modify or adapt responses from a resource. A filter should not have any dependencies on a Web resource for which it is acting as a filter so that it can be composable with more than one type of Web resource.

filter chain

A concatenation of XSLT transformations in which the output of one transformation becomes the input of the next.

form-based authentication

An authentication mechanism in which a Web container provides an application-specific form for logging in. This form of authentication uses Base64 encoding and can expose user names and passwords unless all connections are over SSL.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

general entity

An entity that is referenced as part of an XML document's content, as distinct from a parameter entity, which is referenced in the DTD. A general entity can be a parsed entity or an unparsed entity.

group

An authenticated set of users classified by common traits such as job title or customer profile. Groups are also associated with a set of roles, and every user that is a member of a group inherits all of the roles assigned to that group.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Host element

A representation of a virtual host.

HTML

Hypertext Markup Language. A markup language for hypertext documents on the Internet. HTML enables the embedding of images, sounds, video streams, form fields, references to other objects with URLs and basic text formatting.

HTTP

Hypertext Transfer Protocol. The Internet protocol used to fetch hypertext objects from remote hosts. HTTP messages consist of requests from client to server and responses from server to client.

HTTPS

HTTP layered over the SSL protocol.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

impersonation

An act whereby one entity assumes the identity and privileges of another entity without restrictions and without any indication visible to the recipients of the impersonator's calls that delegation has taken place. Impersonation is a case of simple *delegation*.

initialization parameter

A parameter that initializes the context associated with a servlet.

ISO 3166

The international standard for country codes maintained by the International Organization for Standardization (ISO).

ISV

Independent Software Vendor.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**J2EE™**

See *Java 2 Platform, Enterprise Edition*.

J2ME™

See *Java 2 Platform, Micro Edition*.

J2SE™

See *Java 2 Platform, Standard Edition*.

JAR

Java ARchive. A platform-independent file format that permits many files to be aggregated into one file.

Java™ 2 Platform, Enterprise Edition (J2EE)

An environment for developing and deploying enterprise applications. The J2EE platform consists of a set of services, application programming interfaces (APIs), and protocols that provide the functionality for developing multitiered, Web-based applications.

Java 2 Platform, Micro Edition (J2ME)

A highly optimized Java runtime environment targeting a wide range of consumer products, including pagers, cellular phones, screenphones, digital set-top boxes and car navigation systems.

Java 2 Platform, Standard Edition (J2SE)

The core Java technology platform.

Java API for XML Messaging (JAXM)

An API that provides a standard way to send XML documents over the Internet from the Java platform. It is based on the SOAP 1.1 and SOAP with Attachments specifications, which define a basic framework for exchanging XML messages. JAXM can be extended to work with higher level messaging protocols, such as the one defined in the ebXML (electronic business XML) Message Service Specification, by adding the protocol's functionality on top of SOAP.

Java API for XML Processing (JAXP)

An API for processing XML documents. JAXP leverages the parser standards SAX and DOM so that you can choose to parse your data as a stream

of events or to build a tree-structured representation of it. The latest versions of JAXP also support the XSLT (XML Stylesheet Language Transformations) standard, giving you control over the presentation of the data and enabling you to convert the data to other XML documents or to other formats, such as HTML. JAXP also provides namespace support, allowing you to work with schemas that might otherwise have naming conflicts.

Java API for XML Registries (JAXR)

An API for accessing different kinds of XML registries.

Java API for XML-based RPC (JAX-RPC)

An API for building Web services and clients that use remote procedure calls (RPC) and XML.

Java Naming and Directory Interface™ (JNDI)

An API that provides naming and directory functionality.

Java™ Secure Socket Extension (JSSE)

A set of packages that enable secure Internet communications.

Java™ Transaction API (JTA)

An API that allows applications to access transactions.

Java™ Web Services Developer Pack (Java WSDP)

An environment containing key technologies to simplify building of Web services using the Java 2 Platform.

JavaBeans™ component

A Java class that can be manipulated in a visual builder tool and composed into applications. A JavaBeans component must adhere to certain property and event interface conventions.

JavaMail™

An API for sending and receiving email.

JavaServer Pages™ (JSP™)

An extensible Web technology that uses template data, custom elements, scripting languages, and server-side Java objects to return dynamic content to a client. Typically the template data is HTML or XML elements, and in many cases the client is a Web browser.

JavaServer Pages Standard Tag Library (JSTL)

A tag library that encapsulates core functionality common to many JSP applications. JSTL has support for common, structural tasks such as iteration and conditionals, tags for manipulating XML documents, internationalization and locale-specific formatting tags, and SQL tags. It also introduces a new expression language to simplify page development, and provides an

API for developers to simplify the configuration of JSTL tags and the development of custom tags that conform to JSTL conventions.

JAXR client

A client program that uses the JAXR API to access a business registry via a JAXR provider.

JAXR provider

An implementation of the JAXR API that provides access to a specific registry provider or to a class of registry providers that are based on a common specification.

JDBC™

An API for database-independent connectivity to a wide range of data sources.

JNDI

See *Java Naming and Directory Interface*.

JSP

See *JavaServer Pages*.

JSP action

A JSP element that can act on implicit objects and other server-side objects or can define new scripting variables. Actions follow the XML syntax for elements with a start tag, a body and an end tag; if the body is empty it can also use the empty tag syntax. The tag must use a prefix.

JSP action, custom

An action described in a portable manner by a tag library descriptor and a collection of Java classes and imported into a JSP page by a `taglib` directive. A custom action is invoked when a JSP page uses a custom tag.

JSP action, standard

An action that is defined in the JSP specification and is always available to a JSP file without being imported.

JSP application

A stand-alone Web application, written using the JavaServer Pages technology, that can contain JSP pages, servlets, HTML files, images, applets, and JavaBeans components.

JSP container

A *container* that provides the same services as a *servlet container* and an engine that interprets and processes JSP pages into a servlet.

JSP container, distributed

A JSP container that can run a Web application that is tagged as distributable and is spread across multiple Java virtual machines that might be running on different hosts.

JSP declaration

A JSP scripting element that declares methods, variables, or both in a JSP file.

JSP directive

A JSP element that gives an instruction to the JSP container and is interpreted at translation time.

JSP element

A portion of a JSP page that is recognized by a JSP translator. An element can be a *directive*, an *action*, or a *scripting element*.

JSP expression

A scripting element that contains a valid scripting language expression that is evaluated, converted to a `String`, and placed into the implicit out object.

JSP file

A file that contains a JSP page. In the Servlet 2.2 specification, a JSP file must have a `.jsp` extension.

JSP page

A text-based document using fixed template data and JSP elements that describes how to process a request to create a response.

JSP scripting element

A JSP *declaration*, *scriptlet*, or *expression*, whose tag syntax is defined by the JSP specification, and whose content is written according to the scripting language used in the JSP page. The JSP specification describes the syntax and semantics for the case where the language page attribute is `"java"`.

JSP scriptlet

A JSP scripting element containing any code fragment that is valid in the scripting language used in the JSP page. The JSP specification describes what is a valid scriptlet for the case where the language page attribute is `"java"`.

JSP tag

A piece of text between a left angle bracket and a right angle bracket that is used in a JSP file as part of a JSP element. The tag is distinguishable as markup, as opposed to data, because it is surrounded by angle brackets.

JSP tag library

A collection of custom tags identifying custom actions described via a tag library descriptor and Java classes.

JTA

See *Java Transaction API*.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

life cycle

The framework events of a component's existence. Each type of component has defining events which mark its transition into states where it has varying availability for use. For example, a servlet is created and has its `init` method called by its container prior to invocation of its service method by clients or other servlets who require its functionality. After the call of its `init` method it has the data and readiness for its intended use. The servlet's `destroy` method is called by its container prior to the ending of its existence so that processing associated with winding up may be done, and resources may be released. The `init` and `destroy` methods in this example are *callback methods*.

local subset

That part of the DTD that is defined within the current XML file.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

message-driven bean

An enterprise bean that is an asynchronous message consumer. A message-driven bean has no state for a specific client, but its instance variables may contain state across the handling of client messages. A client accesses a message-driven bean by sending messages to the destination for which the bean is a message listener.

mixed-content model

A DTD specification that defines an element as containing a mixture of text and one more other elements. The specification must start with `#PCDATA`, followed by alternate elements, and must end with the "zero-or-more" asterisk symbol (*).

mutual authentication

An authentication mechanism employed by two parties for the purpose of proving each other's identity to one another.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**namespace**

A standard that lets you specify a unique label to the set of element names defined by a DTD. A document using that DTD can be included in any other document without having a conflict between element names. The elements defined in your DTD are then uniquely identified so that, for example, the parser can tell when an element called <name> should be interpreted according to your DTD, rather than using the definition for an element called name in a different DTD.

naming context

A set of associations between unique, atomic, people-friendly identifiers and objects.

naming environment

A mechanism that allows a component to be customized without the need to access or change the component's source code. A container implements the component's naming environment, and provides it to the component as a *JNDI naming context*. Each component names and accesses its environment entries using the `java:comp/env` JNDI context. The environment entries are declaratively specified in the component's deployment descriptor.

normalization

The process of removing redundancy by modularizing, as with subroutines, and of removing superfluous differences by reducing them to a common denominator. For example, line endings from different systems are normalized by reducing them to a single NL, and multiple whitespace characters are normalized to one space.

North American Industry Classification System (NAICS)

A system for classifying business establishments based on the processes they use to produce goods or services.

notation

A mechanism for defining a data format for a non-XML document referenced as an unparsed entity. This is a holdover from SGML that creaks a bit. The newer standard is to use MIME datatypes and namespaces to prevent naming conflicts.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**OASIS**

Organization for the Advancement of Structured Information Standards. Their home site is <http://www.oasis-open.org/>. The DTD repository they sponsor is at <http://www.XML.org>.

one-way messaging

A method of transmitting messages without having to block until a response is received.

OS principal

A principal native to the operating system on which the Web services platform is executing.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**parameter entity**

An entity that consists of DTD specifications, as distinct from a general entity. A parameter entity defined in the DTD can then be referenced at other points, in order to prevent having to recode the definition at each location it is used.

parsed entity

A general entity that contains XML, and which is therefore parsed when inserted into the XML document, as opposed to an unparsed entity.

parser

A module that reads in XML data from an input source and breaks it up into chunks so that your program knows when it is working with a tag, an attribute, or element data. A nonvalidating parser ensures that the XML data is well formed, but does not verify that it is valid. See also: validating parser.

principal

The identity assigned to a user as a result of authentication.

privilege

A security attribute that does not have the property of uniqueness and that may be shared by many principals.

processing instruction

Information contained in an XML structure that is intended to be interpreted by a specific application.

programmatic security

Security decisions that are made by security-aware applications. Programmatic security is useful when declarative security alone is not sufficient to express the security model of an application.

prolog

The part of an XML document that precedes the XML data. The prolog includes the declaration and an optional DTD.

public key certificate

Used in client-certificate authentication to enable the server, and optionally the client, to authenticate each other. The public key certificate is a digital equivalent of a passport. It is issued by a trusted organization, called a certificate authority (CA), and provides identification for the bearer.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

RDF

Resource Description Framework. A standard for defining the kind of data that an XML file contains. Such information could help ensure semantic integrity, for example by helping to make sure that a date is treated as a date, rather than simply as text.

RDF schema

A standard for specifying consistency rules that apply to the specifications contained in an RDF.

reference

See entity reference

realm

See *security policy domain*. Also, a string, passed as part of an HTTP request during *basic authentication*, that defines a protection space. The protected resources on a server can be partitioned into a set of protection spaces, each with its own authentication scheme and/or authorization database.

registry

An infrastructure that enables the building, deployment and discovery of Web services. It is a neutral third party that facilitates dynamic and loosely coupled business-to-business (B2B) interactions.

registry provider

An implementation of a business registry that conforms to a specification for XML registries.

request-response messaging

A method of messaging that includes blocking until a response is received.

resource manager

Provides access to a set of shared resources. A resource manager participates in transactions that are externally controlled and coordinated by a transaction manager. A resource manager is typically in different address space or on a different machine from the clients that access it. Note: An *enterprise information system* is referred to as resource manager when it is mentioned in the context of resource and transaction management.

resource manager connection

An object that represents a session with a resource manager.

resource manager connection factory

An object used for creating a resource manager connection.

role (security)

An abstract logical grouping of users that is defined by the Application Assembler. When an application is deployed, the roles are mapped to security identities, such as *principals* or *groups*, in the operational environment.

A role can be compared to a key that can open a lock. Many people might have a copy of the key, and the lock doesn't care who you are, only that you have the right key.

role mapping

The process of associating the groups and/or principals recognized by the container to security roles specified in the *deployment descriptor*. Security roles have to be mapped before a component is installed in the server.

rollback

The point in a transaction when all updates to any resources involved in the transaction are reversed.

root

The outermost element in an XML document. The element that contains all other elements.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**SAX**

Simple API for XML. An event-driven interface in which the parser invokes one of several methods supplied by the caller when a parsing event occurs.

Events include recognizing an XML tag, finding an error, encountering a reference to an external entity, or processing a DTD specification.

schema

A database-inspired method for specifying constraints on XML documents using an XML-based language. Schemas address deficiencies in DTDs, such as the inability to put constraints on the kinds of data that can occur in a particular field. Since schemas are founded on XML, they are hierarchical, so it is easier to create an unambiguous specification, and possible to determine the scope over which a comment is meant to apply.

Secure Socket Layer (SSL)

A technology that allows Web browsers and Web servers to communicate over a secured connection.

security attributes

A set of properties associated with a principal. Security attributes can be associated with a principal by an authentication protocol or by a Java WSDP Product Provider.

security constraint

Determines who is authorized to access a Web resource collection.

security context

An object that encapsulates the shared state information regarding security between two entities.

security permission

A mechanism, defined by J2SE, to express the programming restrictions imposed on component developers.

security policy domain

A scope over which security policies are defined and enforced by a security administrator. A security policy domain has a collection of users (or principals), uses a well defined authentication protocol(s) for authenticating users (or principals), and may have groups to simplify setting of security policies.

security role

See *role (security)*.

security technology domain

A scope over which the same security mechanism is used to enforce a security policy. Multiple security policy domains can exist within a single technology domain.

server certificate

Used with HTTPS protocol to authenticate Web applications. The certificate can be self-signed or approved by a Certificate Authority (CA).

server principal

The OS principal that the server is executing as.

service element

A representation of the combination of one or more Connector components that share a single engine component for processing incoming requests.

servlet

A Java program that extends the functionality of a Web server, generating dynamic content and interacting with Web applications using a request-response paradigm.

servlet container

A *container* that provides the network services over which requests and responses are sent, decodes requests, and formats responses. All servlet containers must support HTTP as a protocol for requests and responses, but may also support additional request-response protocols such as HTTPS.

servlet container, distributed

A servlet container that can run a Web application that is tagged as distributable and that executes across multiple Java virtual machines running on the same host or on different hosts.

servlet context

An object that contains a servlet's view of the Web application within which the servlet is running. Using the context, a servlet can log events, obtain URL references to resources, and set and store attributes that other servlets in the context can use.

servlet mapping

Defines an association between a URL pattern and a servlet. The mapping is used to map requests to servlets.

session

An object used by a servlet to track a user's interaction with a Web application across multiple HTTP requests.

session bean

An enterprise bean that is created by a client and that usually exists only for the duration of a single client-server session. A session bean performs operations, such as calculations or accessing a database, for the client. Although a session bean may be transactional, it is not recoverable should a system crash occur. Session bean objects can be either stateless or can maintain conversational state across methods and transactions. If a session bean maintains state, then the EJB container manages this state if the object must be removed from memory. However, the session bean object itself must manage its own persistent data.

SGML

Standard Generalized Markup Language. The parent of both HTML and XML. However, while HTML shares SGML's propensity for embedding presentation information in the markup, XML is a standard that allows information content to be totally separated from the mechanisms for rendering that content.

SOAP

Simple Object Access Protocol

SOAP with Attachments API for Java (SAAJ)

The basic package for SOAP messaging which contains the API for creating and populating a SOAP message.

SSL

Secure Socket Layer. A security protocol that provides privacy over the Internet. The protocol allows client-server applications to communicate in a way that cannot be eavesdropped or tampered with. Servers are always authenticated and clients are optionally authenticated.

SQL

Structured Query Language. The standardized relational database language for defining database objects and manipulating data.

SQL/J

A set of standards that includes specifications for embedding SQL statements in methods in the Java programming language and specifications for calling Java static methods as SQL stored procedures and user-defined functions. An SQL checker can detect errors in static SQL statements at program development time, rather than at execution time as with a JDBC driver.

standalone client

A client that does not use a messaging provider and does not run in a container.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**tag**

A piece of text that describes a unit of data, or element, in XML. The tag is distinguishable as markup, as opposed to data, because it is surrounded by angle brackets (< and >). To treat such markup syntax as data, you use an entity reference or a CDATA section.

template

A set of formatting instructions that apply to the nodes selected by an XPATH expression.

transaction

An atomic unit of work that modifies data. A transaction encloses one or more program statements, all of which either complete or roll back. Transactions enable multiple users to access the same data concurrently.

transaction isolation level

The degree to which the intermediate state of the data being modified by a transaction is visible to other concurrent transactions and data being modified by other transactions is visible to it.

transaction manager

Provides the services and management functions required to support transaction demarcation, transactional resource management, synchronization, and transaction context propagation.

translet

Pre-compiled version of a transformation.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Unicode

A standard defined by the Unicode Consortium that uses a 16-bit code page which maps digits to characters in languages around the world. Because 16 bits covers 32,768 codes, Unicode is large enough to include all the world's languages, with the exception of ideographic languages that have a different character for every concept, like Chinese. For more info, see <http://www.unicode.org/>.

Universal Description, Discovery, and Integration (UDDI) project

An industry initiative to create a platform-independent, open framework for describing services, discovering businesses, and integrating business services using the Internet, as well as a registry. It is being developed by a vendor consortium.

Universal Standard Products and Services Classification (UNSPSC)

A schema that classifies and identifies commodities. It is used in sell side and buy side catalogs and as a standardized account code in analyzing expenditure.

unparsed entity

A general entity that contains something other than XML. By its nature, an unparsed entity contains binary data.

URI

Uniform Resource Identifier. A globally unique identifier for an abstract or physical resource. A *URL* is a kind of URI that specifies the retrieval protocol (http or https for Web applications) and physical location of a resource (host name and host-relative path). A *URN* is another type of URI.

URL

Uniform Resource Locator. A standard for writing a textual reference to an arbitrary piece of data in the World Wide Web. A URL looks like `protocol://host/localinfo` where `protocol` specifies a protocol for fetching the object (such as HTTP or FTP), `host` specifies the Internet name of the targeted host, and `localinfo` is a string (often a file name) passed to the protocol handler on the remote host.

URL path

The part of a URL passed by an HTTP request to invoke a servlet. A URL path consists of the Context Path + Servlet Path + Path Info, where

- Context Path is the path prefix associated with a servlet context that this servlet is a part of. If this context is the default context rooted at the base of the Web server's URL namespace, the path prefix will be an empty string. Otherwise, the path prefix starts with a / character but does not end with a / character.
- Servlet Path is the path section that directly corresponds to the mapping which activated this request. This path starts with a / character.
- Path Info is the part of the request path that is not part of the Context Path or the Servlet Path.

URN

Uniform Resource Name. A unique identifier that identifies an entity, but doesn't tell where it is located. A system can use a URN to look up an entity locally before trying to find it on the Web. It also allows the Web location to change, while still allowing the entity to be found.

user (security)

An individual (or application program) identity that has been authenticated. A user can have a set of roles associated with that identity, which entitles them to access all resources protected by those roles.

user data constraint

Indicates how data between a client and a Web container should be protected. The protection can be the prevention of tampering with the data or prevention of eavesdropping on the data.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

valid

A valid XML document, in addition to being well formed, conforms to all the constraints imposed by a DTD. It does not contain any tags that are not

permitted by the DTD, and the order of the tags conforms to the DTD's specifications.

validating parser

A parser that ensures that an XML document is valid, as well as well-formed. See also: parser.

virtual host

Multiple “hosts + domain names” mapped to a single IP.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

W3C

World Wide Web Consortium. The international body that governs Internet standards.

WAR file

Web application archive. A JAR archive that contains a Web module.

warning

A SAX parser warning is generated when the document's DTD contains duplicate definitions, and similar situations that are not necessarily an error, but which the document author might like to know about, since they could be. See also: fatal error, error.

Web application

An application written for the Internet, including those built with Java technologies such as JavaServer Pages and servlets, as well as those built with non-Java technologies such as CGI and Perl.

Web Application Archive (WAR)

A hierarchy of directories and files in a standard Web application format, contained in a packed file with an extension .war.

Web application, distributable

A Web application that uses Java WSDP technology written so that it can be deployed in a Web container distributed across multiple Java virtual machines running on the same host or different hosts. The deployment descriptor for such an application uses the distributable element.

Web component

A component that provides services in response to requests; either a *servlet* or a *JSP page*.

Web container

A *container* that implements the Web component contract of the J2EE architecture. This contract specifies a runtime environment for Web components that includes security, concurrency, life cycle management, transaction,

deployment, and other services. A Web container provides the same services as a *JSP container* and a federated view of the J2EE platform APIs. A Web container is provided by a *Web server*.

Web container, distributed

A Web container that can run a Web application that is tagged as distributable and that executes across multiple Java virtual machines running on the same host or on different hosts.

Web module

A unit that consists of one or more Web components, other resources, and a Web deployment descriptor.

Web resource

A static or dynamic object contained in a Web application archive that can be referenced by a URL.

Web resource collection

A list of URL patterns and HTTP methods that describe a set of resources to be protected.

Web server

Software that provides services to access the Internet, an intranet, or an extranet. A Web server hosts Web sites, provides support for HTTP and other protocols, and executes server-side programs (such as CGI scripts or servlets) that perform certain functions. In the J2EE architecture, a Web server provides services to a *Web container*. For example, a Web container typically relies on a Web server to provide HTTP message handling. The J2EE architecture assumes that a Web container is hosted by a Web server from the same vendor, so does not specify the contract between these two entities. A Web server may host one or more Web containers.

Web service

An application that exists in a distributed environment, such as the Internet. A Web service accepts a request, performs its function based on the request, and returns a response. The request and the response can be part of the same operation, or they can occur separately, in which case the consumer does not need to wait for a response. Both the request and the response usually take the form of XML, a portable data-interchange format, and are delivered over a wire protocol, such as HTTP.

well-formed

An XML document that is syntactically correct. It does not have any angle brackets that are not part of tags, all tags have an ending tag or are themselves self-ending, and all tags are fully nested. Knowing that a document is well formed makes it possible to process it. A well-formed document may

not be valid however. To determine that, you need a *validating parser* and a *DTD*.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Xalan

An interpreting version of XSLT.

XHTML

An XML lookalike for *HTML* defined by one of several XHTML DTDs. To use XHTML for everything would of course defeat the purpose of XML, since the idea of XML is to identify information content, not just tell how to display it. You can reference it in a DTD, which allows you to say, for example, that the text in an element can contain `` and `` tags, rather than being limited to plain text.

XLink

The part of the XLL specification that is concerned with specifying links between documents.

XLL

The XML Link Language specification, consisting of *XLink* and *XPointer*.

XML

Extensible Markup Language. A markup language that allows you to define the tags (markup) needed to identify the content, data, and text, in XML documents. It differs from *HTML* the markup language most often used to present information on the internet. HTML has fixed tags that deal mainly with style or presentation. An XML document must undergo a transformation into a language with style tags under the control of a stylesheet before it can be presented by a browser or other presentation mechanism. Two types of style sheets used with XML are *CSS* and *XSL*. Typically, XML is transformed into HTML for presentation. Although tags may be defined as needed in the generation of an XML document, a *DTD* may be used to define the elements allowed in a particular type of document. A document may be compared with the rules in the DTD to determine its validity and to locate particular elements in the document. Web services application's deployment descriptors are expressed in XML with DTDs defining allowed elements. Programs for processing XML documents use *SAX* or *DOM* APIs.

XML registry

See registry.

XML Schema

The W3C schema specification for XML documents.

XPath

See XSL.

XPointer

The part of the XLL specification that is concerned with identifying sections of documents so that they can referenced in links or included in other documents.

XSL

Extensible Stylesheet Language. Extensible Stylesheet Language. An important standard that achieves several goals. XSL lets you:

- a.Specify an addressing mechanism, so you can identify the parts of an XML file that a transformation applies to. (XPath)
- b.Specify tag conversions, so you convert XML data into a different formats. (XSLT)
- c.Specify display characteristics, such page sizes, margins, and font heights and widths, as well as the flow objects on each page. Information fills in one area of a page and then automatically flows to the next object when that area fills up. That allows you to wrap text around pictures, for example, or to continue a newsletter article on a different page. (XML-FO)

XSL-FO

A subcomponent of XSL used for describing font sizes, page layouts, and how information “flows” from one page to another.

XSLT

XSL Transformation. An XML file that controls the transformation of an XML document into another XML document or HTML. The target document often will have presentation related tags dictating how it will be rendered by a browser or other presentation mechanism. XSLT was formerly part of XSL, which also included a tag language of style flow objects.

XSLTC

A compiling version of XSLT.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

About the Authors

Java API for XML Processing

Eric Armstrong has been programming and writing professionally since before there were personal computers. His production experience includes artificial intelligence (AI) programs, system libraries, real-time programs, and business applications in a variety of languages. He works as a consultant at Sun's Java Software division in the Bay Area, and he is a contributor to JavaWorld. He wrote *The JBuilder2 Bible*, as well as Sun's Java XML programming tutorial. For a time, Eric was involved in efforts to design next-generation collaborative discussion/decision systems. His learn-by-ear, see-the-fingering music teaching program is currently on hold while he finishes a weight training book. His Web site is <http://www.treelight.com>.

Web Applications and Technology

Stephanie Bodoff is a staff writer at Sun Microsystems. In previous positions she worked as a software engineer on distributed computing and telecommunications systems and object-oriented software development methods. Since her conversion to technical writing, Stephanie has documented object-oriented databases, application servers, and enterprise application development methods. She is a co-author of *The J2EE Tutorial*, *Designing Enterprise Applications with the Java™ 2 Platform, Enterprise Edition*, and *Object-Oriented Software Development: The Fusion Method*.

Java API for XML Messaging, Introduction to Web Services

Maydene Fisher has documented various Java APIs at Sun Microsystems for the last five years. She authored two books on the JDBC API, *JDBC™ Database Access with Java: A Tutorial and Annotated Reference* and *JDBC™ API Tutorial and Reference, Second Edition: Universal Data Access for the Java™ 2 Platform*. Before joining Sun, she helped document the object-oriented programming language ScriptX at Kaleida Labs and worked on Wall Street, where she wrote developer and user manuals for complex financial

computer models written in C++. In previous lives, she has been an English teacher, a shopkeeper in Mendocino, and a financial planner.

Java API for RPC-based XML

Dale Green is a staff writer with Sun Microsystems, where he documents the J2EE platform and the Java API for RPC-based XML. In previous positions he programmed business applications, designed databases, taught technical classes, and documented RDBMS products. He wrote the Internationalization and Reflection trails for the *Java Tutorial Continued*, and co-authored *The J2EE Tutorial*.

Java API for XML Registries, Java WSDP Registry Server

Kim Haase is a staff writer with Sun Microsystems, where she documents the J2EE platform. In previous positions she has documented compilers, debuggers, and floating-point programming. She currently writes about the Java Message Service and the Java API for XML Registries. She is a co-author of *Java™ Message Service API Tutorial and Reference*.

Index

A

- actors 300
- addChildElement method 305
- addClassifications method 363
- addExternalLink method 368
- addServiceBindings method 364
- addServices method 364
- addTextNode method 305
- ANY 214
- applet containers 13
- applets 7, 9
- applications
 - standalone 324
- archiving 186
- attachment part
 - populating 261
- AttachmentPart class 294
- AttachmentPart object 315
 - creating 316
 - headers 316
- attachments 294
 - adding 315
- attributes 179, 201, 300, 307
 - defining in DTD 216
 - encoding 181
 - mustUnderstand 300
 - standalone 181
 - types 217
 - version 181

- attribute-specification parameters 218
- authentication
 - for XML registries 361

B

- binding 186
- binding templates
 - adding to an organization with JAXR 363
 - finding with JAXR 360
- body
 - adding content 305
- BufferedReader 60
- businesses
 - contacts 361
 - creating with JAXR 361
 - finding
 - by name with JAXR 357, 376
 - using WSDL documents with JAXR 377
 - finding by classification with JAXR 358, 377
 - keys 361, 365
 - registering 263
 - removing
 - with JAXR 365, 377

- saving
 - with JAXR 364, 376
- BusinessLifeCycleManager interface 350, 356, 361
 - See also* LifeCycleManager interface
- BusinessQueryManager interface 350, 356
- C**
- call method 308, 340
- capability levels 348
- CBL 199
- CDATA
 - versus PCDATA 212
- classes
 - AttachmentPart 294
 - ConnectionFactory 352
 - javax.xml.soap.SOAPConnection 296
 - SOAPEnvelope 293
 - SOAPFactory 305
 - SOAPMessage 293
 - SOAPPart 293
- classification schemes
 - finding with JAXR 363
 - ISO 3166 357
 - NAICS 357, 377
 - postal address 367, 378
 - publishing 367, 378
 - removing 380
 - UNSPSC 357
 - user-defined 366
- classifications
 - creating with JAXR 363
- client
 - application
 - for a Web service 254
 - JAXR 349
 - implementing 351
 - querying a registry 356
 - standalone 257, 258, 301, 308
- Collection interface 271
- com.sun.xml.registry.http.proxyHost connection property 354
- com.sun.xml.registry.http.proxyPort connection property 354
- com.sun.xml.registry.http.proxyHost connection property 354
- com.sun.xml.registry.http.proxyPassword connection property 355
- com.sun.xml.registry.http.proxyPort connection property 354
- com.sun.xml.registry.http.proxyUserName connection property 355
- com.sun.xml.registry.userTaxonomyFileNames system property 369
- com.sun.xml.registry.useSOAP connection property 355
- comment 200
- concepts
 - in user-defined classification schemes 366
 - using to create classifications with JAXR 363
- conditional sections 227
- connection 295
 - close 308

- creating 352
- getting 258
- point-to-point 258, 296, 303
- setting properties 352
- to the messaging provider 258
- connection factory
 - JAXR
 - creating 352
- Connection interface 349, 352
- connection properties
 - `com.sun.xml.registry.http.proxyHost` 354
 - `com.sun.xml.registry.http.proxyPort` 354
 - `com.sun.xml.registry.http.proxyHost` 354
 - `com.sun.xml.registry.http.proxyPassword` 355
 - `com.sun.xml.registry.http.proxyPort` 354
 - `com.sun.xml.registry.http.proxyUserName` 355
 - `com.sun.xml.registry.useSOAP` 355
- examples 352
- `javax.xml.registry.lifecycleManagerURL` 353
- `javax.xml.registry.postalAddressScheme` 354, 370
- `javax.xml.registry.queryManagerURL` 353
- `javax.xml.registry.security.authentication.Method` 354
- `javax.xml.registry.seman-`
 - `ticEquivalences` 353, 370
 - `javax.xml.registry.udi.maxRows` 354
- ConnectionFactory class 352
- connectors
 - See J2EE Connector technology
- container 250
- containers 11
 - See also
 - applet containers
 - EJB containers
 - J2EE application clients, containers
 - Web containers
 - services 12
- context roots 30
- country codes
 - ISO 3166 357
- `createAttachmentPart` method 318
- `createClassification` method 363, 367
- `createClassificationScheme` method 367
- `createExternalLink` method 367
- `createOrganization` method 362
- `createPostalAddress` method 371
- `createService` method 364
- `createServiceBinding` method 364
- custom tags 111
 - attributes 120
 - validation 129
- bodies 120
- cooperating 121
- defining 122

- examples 142
 - scripting variables
 - defining 121
 - providing information about 133, 135
 - Struts tag library 115
 - tag handlers 125
 - defining scripting variables 132
 - methods 125
 - simple tag 126
 - with attributes 127
 - with bodies 130
 - tag library descriptors
 - See tag library descriptors
 - tutorial-template tag library 115
- cxml 198
- D**
- data
 - element 216
 - normalizing 236
 - processing 185
 - databases
 - clients 7
 - EIS tier 5
 - DDP
 - declaration 181, 200
 - defining text 212
 - deleteOrganizations method 365
 - deployer role 17
 - deployment 254
 - deployment descriptors 14
 - web application 26
 - destroy 79
 - Detail object 321
 - DetailEntry object 321
 - development roles 14
 - DII 282
 - doAfterBody 130
 - document
 - element 216
 - Document Object Model
 - See DOM
 - Document-Driven Programming
 - See DDP
 - doEndTag 126
 - doFilter 65, 66, 71, 72
 - doGet 59
 - doInitBody 130
 - DOM 189, 242, 245, 267
 - transforming to an XML Document 248
 - dom4j 189
 - doPost 59
 - doStartTag 126
 - downloading
 - J2EE SDK ix
 - tutorial ix
 - DrawML 198
 - DTD 181, 190, 192
 - defining attributes 216
 - defining entities 219
 - defining namespaces 230
 - factoring out 237
 - industry-standard 233
 - limitations 213
 - normalizing 237
 - dynamic invocation interface
 - See DII
 - dynamic proxies 280

E

- EAR files 13
- ease of use 251
- ebXML 198, 256, 263
 - Message Service Specification 256
 - profile 299, 311
 - registries 348, 349
- EIS 11
- EJB containers 13
- electronic business XML
 - See* ebXML
- element 201
 - empty 203
 - nested 202
 - qualifiers 212
 - root 200
- eliminating redundancies 236
- EMPTY 214
- encoding 181
- endpoint 308
- enterprise beans 9
 - development role 16
 - types 10
- Enterprise Information Systems
 - See* EIS
- entities 181
 - defining in DTD 219
 - external 236
 - included "in line" 183
 - parameter 225
 - parsed 222
 - predefined 207
 - reference 236
 - referencing binary 223
 - referencing external 221
 - unparsed 222
 - useful 220

examples

- downloading ix
- location ix

exceptions

- mapping to web resources 32
- web components 32

Extensible Markup Language

- See* XML

F**fault**

- code 320
- retrieving information 322
- string 320

Filter 65

- filter chains 65, 71

filters 64

- defining 65
- mapping to Web components 70
- mapping to Web resources 70, 71, 72
- overriding request methods 67
- overriding response methods 67
- response wrapper 67

findClassificationSchemeByName

- method 363, 367

findConcepts method 359**findOrganization method 357****forward 74****fully qualified 305****G****GenericServlet 48**

- getChildElements method 331

- getParameter 60
- getRequestDispatcher 72
- getServletContext 75
- getSession 76

H

- headers 300
 - adding content 312
 - Content-Id 316
 - Content-Location 316
 - Content-Type 316
- HTML 178
- HTTP 269
 - setting proxies 354
- HTTP protocol 427
- HTTP requests 60, 428
 - methods 428
 - query strings 61
 - See also requests
 - URLs 60
- HTTP responses 62, 428
 - See also responses
 - status codes 32, 428
 - mapping to web resources 32
- HttpServlet 48
- HttpServletRequest 60
- HttpServletResponse 62
- HttpSession 76

I

- ICE 198
- include 73, 98
- information model
 - JAXR 348, 349
- init 59

- instructions
 - processing 182, 204
- interfaces
 - BusinessLifeCycleManager 350, 356, 361
 - BusinessQueryManager 350, 356
 - Collection 271
 - Connection 349, 352
 - javax.xml.messaging.ProviderConnection 296
 - Organization 361
 - RegistryObject 349
 - RegistryService 349, 356
 - SOAPEnvelope 307
- interoperability 241, 251
- invalidate 78
- ISO 3166 country codes 357
- isThreadSafe 94
- isValid 129

J

- J2EE application clients 7
 - containers 13
- J2EE applications 5
 - assembler role 16
 - tiers 5
- J2EE clients 7
 - application clients 7
 - See also J2EE application clients
 - Web clients 7
 - See also Web clients
- web clients 25
 - See also web clients
- Web clients versus J2EE application clients 8

- J2EE components
 - defined 6
 - types 6
- J2EE Connector technology
 - architecture version 22
- J2EE modules 14
- J2EE platform 1, 5
- J2EE SDK
 - downloading ix
- J2EE server 13
- J2EE Technology in Practice* 23
- J2SE SDK 270
- JAAS 22
- JAF 20
- JAR files
 - See also
 - EJB JAR files
- Java 2 Platform, Enterprise Edition
 - See J2EE
- Java API for XML Messaging
 - See JAXM
- Java API for XML Processing
 - See JAXP
 - See JAXP
- Java API for XML Registries
 - See JAXR
- Java API for XML-based RPC
 - See JAX-RPC
- Java Authentication and Authorization Service
 - See JAAS
- Java Message Service
 - See JMS
- Java Naming and Directory Interface
 - See JNDI
- Java Naming and Directory Interface (JNDI) API 310
- Java Naming and Directory Interface API
 - See JNDI
- Java Servlet technology 18
 - See also servlets
- Java Transaction API
 - See JTA
- JavaBeans Activation Framework
 - See JAF
- JavaBeans components 8, 272
 - benefits of using 105
 - creating in JSP pages 106
 - design conventions 103
 - in WAR files 28
 - methods 104
 - properties 103
 - retrieving in JSP pages 109
 - setting in JSP pages 106
 - using in JSP pages 105
- JavaMail API 20
- JavaServer Pages
 - See JSP
- JavaServer Pages (JSP) technology 18
 - See also JSP pages
- `javax.activation.DataHandler` object 317
- `javax.servlet` 48
- `javax.servlet.http` 48
- `javax.xml.messaging` package 290
- `javax.xml.messaging.ProviderConnection` interface 296
- `javax.xml.registry` package 349
- `javax.xml.registry.infomodel` package 349
- `javax.xml.registry.LifecycleManagerURL` connection property

- 353
- `javax.xml.registry postalAddressScheme` connection property 354, 370
- `javax.xml.registry.queryManagerURL` connection property 353
- `javax.xml.registry.security.authenticationMethod` connection property 354
- `javax.xml.registry.semanticEquivalences` connection property 353, 370
- `javax.xml.registry.udi.maxRows` connection property 354
- `javax.xml.soap` package 290
- `javax.xml.soap.SOAPConnection` class 296
- `javax.xml.soap.SOAPConnection.call` method 297
- `javax.xml.transform` package 248
- `javax.xml.transform.Source` object 314
- JAXM 241, 252, 256, 289
 - 1.0 specification 290
 - API 290
 - Javadoc documentation 299
- JAXM 1.1
 - specification 290
- JAXP 20, 241, 314
- JAXR 241, 252, 263, 347
 - adding
 - classifications 363
 - service bindings 363
 - services 363
 - architecture 349
 - capability levels 348
 - clients 349
 - implementing 351
 - submitting data to a registry 361
 - creating
 - connections 352
 - defining taxonomies 366
 - definition 348
 - establishing security credentials 361
 - finding classification schemes 363
 - information model 348
 - organizations
 - creating 361
 - removing 365
 - saving 364
 - provider 349
 - querying a registry 356
 - specification 348
 - specifying postal addresses 369
 - submitting data to a registry 361
- JAX-RPC 241, 250
 - defined 269
- JavaBeans components 272
- overview 250
- specification 286
- supported types 270
- JDBC API 18
- JDOM 189
- JMS 19
 - tutorial 11
- JNDI 19, 259
- JSP 267
- JSP declarations 95
- JSP expressions 98

- JSP pages 83
 - compilation 88
 - errors 89
 - creating and using objects 94
 - creating dynamic content 92
 - creating static content 92
 - custom tags
 - See custom tags
 - declarations
 - See JSP declarations
 - eliminating scripting 111
 - error page 90
 - examples 27, 85, 86, 114, 149, 150
 - execution 90
 - expressions
 - See JSP expressions
 - finalization 91
 - forwarding to an error page 90
 - forwarding to other Web components 100
 - implicit objects 92
 - importing classes and packages 95
 - importing tag libraries 118
 - including applets or JavaBeans components 100
 - including other Web resources 98
 - initialization 91
 - JavaBeans components
 - creating 106
 - retrieving properties 109
 - setting properties 106
 - from constants 107
 - from request parameters 107
 - from runtime expressions 108
 - using 105
 - life cycle 88
 - scripting elements
 - See JSP scripting elements
 - scriptlets
 - See JSP scriptlets
 - setting buffer size 90
 - shared objects 94
 - specifying scripting language 95
 - translation 88, 89
 - enforcing constraints for custom tag attributes 129
 - errors 89
 - JSP scripting elements 95
 - JSP scriptlets 96
 - drawbacks 111
 - JSP tag libraries 112
 - jsp:fallback 101
 - jsp:forward 100
 - jsp:getProperty 109
 - jsp:include 99
 - jsp:param 100, 101
 - jsp:plugin 100
 - jsp:setProperty 106
 - jspDestroy 91
 - jspInit 91
 - JTA 19
- L**
- linking
 - XML 194
 - listener classes 52
 - defining 52
 - examples 52

listener interfaces 52

local

 name 313

 provider 339

M

MathML 197

message

 accessing elements 304

 creating 259, 303, 311

 getting the content 308

 populating the attachment part
 261

 populating the SOAP part 260

 sending 262, 315

MessageFactory object 259, 311

 getting 311

messaging

 one-way 257, 297

 provider 256, 297–??, 309

 getting a connection 258

 when to use 300

 request-response 257, 296

methods

 addChildElement 305

 addClassifications 363

 addExternalLink 368

 addServiceBindings 364

 addServices 364

 addTextNode 305

 call 308, 340

 createAttachmentPart 318

 createClassification 363,
 367

 createClassificationScheme
 367

 createExternalLink 367

 createOrganization 362

 createPostalAddress 371

 createService 364

 createServiceBinding 364

 deleteOrganizations 365

 findClassificationScheme-
 ByName 363, 367

 findConcepts 359

 findOrganization 357

 getChildElements 331

 javax.xml.soap.SOAPConnec-
 tion.call 297

 Node.getValue 308

 ProviderConnection.send
 297, 298, 315

 saveOrganizations 364

 setContent 316

 setNamespaceAware 247

 setPostalAddresses 371

 SOAPConnection.call 339

 SOAPMessage.getAttachments
 318

 SOAPPart.setContent 314

MIME

 data 223

 header 294

mixed-content model 213

mustUnderstand attribute 300

N

NAICS 377

 using to find organizations 358

name

 local 313

Name object 305, 331

namespaces 191, 247, 305
 declaration 247

- defining a prefix 231
- defining in DTD 230
- fully-qualified 307, 313
- prefix 249, 307
- referencing 231
- support 242
- using 230
- nested elements 212
- Node.getValue method 308
- nodes 292
- normalizing
 - data 236
 - DTDs 237
- North American Industry Classification System
 - See* NAICS

O

- OASIS 233, 241
- objects
 - AttachmentPart 315
 - Detail 321
 - DetailEntry 321
 - javax.activation.DataHandler 317
 - javax.xml.transform.Source 314
 - MessageFactory 259, 311
 - Name 305, 331
 - ProviderConnection 258, 297, 310
 - SOAPBody 257, 293, 305, 307, 313
 - SOAPBodyElement 305, 307, 313, 332
 - SOAPConnection 258, 296, 302, 308

- SOAPElement 306, 332
- SOAPFault 319, 336
- SOAPHeader 293, 305, 312
- SOAPHeaderElement 312
- SOAPMessage 257, 304, 311
- SOAPPart 294, 306, 314
- one-way messaging 257, 290, 297
- Organization for the Advancement of Structured Information Standards
 - See* OASIS
- Organization interface 361
- organizations
 - creating with JAXR 361
 - finding
 - by classification 358, 377
 - by name 357, 376
 - using WSDL documents 377
 - keys 361, 365
 - primary contacts 361
 - publishing 376
 - removing 377
 - with JAXR 365
 - saving
 - with JAXR 364

P

- packages
 - javax.xml.messaging 290
 - javax.xml.registry 349
 - javax.xml.registry.infomodel 349
 - javax.xml.soap 290
 - javax.xml.transform 248
- packaging 254
- parameter entity 225

- parsed
 - character data 212
 - entity 222
- PCDATA 212
 - versus CDATA 212
- pluggability layer 242
- point-to-point connection 303
- postal addresses
 - retrieving 371, 378
 - specifying 369, 378
- prerequisites viii
- printing the tutorial xi
- PrintWriter 62
- processing
 - data 185
 - instructions 182, 204
- profiles 259, 299
 - ebXML 299, 311
 - implementations 312
 - Javadoc documentation 299
 - SOAP-RP 299
- provider
 - JAXR 349
- ProviderConnection object 258, 297, 310
- ProviderConnection.send method 297, 298, 315
- proxies 269, 277
 - HTTP, setting 354

R

- RDF 196
 - schema 196
- registering businesses 263
- registries
 - definition 348
 - ebXML 348, 349

- getting access to public UDDI registries 351
- private 349
- querying 356
- searching 264
- submitting data 361
- UDDI 348
- UDDI Server Registry 349
- using public and private 373
- registry objects 349
 - retrieving 381
- Registry Server
 - See* J2EE SDK Registry Server
- RegistryObject interface 349
- RegistryService interface 349, 356
- RELAX NG 193
- release 131
- remote method invocation 255
- remote procedure call
 - See* RPC
- remote procedure calls 269
- RequestDispatcher 72
- request-response messaging 257, 290, 296, 394, 403
- requests 60
 - appending parameters 100
 - customizing 67
 - getting information from 60
 - retrieving a locale 39
 - See also* HTTP requests
- required software ix
- resource bundles 38
- responses 62
 - buffering output 62
 - customizing 67
 - See also* HTTP responses
 - setting headers 59

- roles
 - development
 - See development roles
- root
 - element 200
- RPC 251, 269
- S**
- SAAJ 292
- SAAJ 1.1
 - API 290, 305
 - specification 290, 297
- sample programs
 - JAXR 371
 - compiling 376
 - editing properties file 373
 - setting classpath 375
- saveOrganizations method 364
- SAX 189, 242, 267
- schema 192
 - RDF 196
 - XML 193
- Schematron 194
- searching registries 264
- security
 - credentials for XML registries 361
- service bindings
 - adding to an organization with JAXR 363
 - finding with JAXR 360
- service endpoint 255
- services
 - adding to an organization with JAXR 363
 - finding with JAXR 360
- Servlet 48
- ServletContext 75
- ServletInputStream 60
- ServletOutputStream 62
- ServletRequest 60
- ServletResponse 62
- servlets 47
 - binary data
 - reading 60
 - writing 62
 - character data
 - reading 60
 - writing 62
 - examples 27
 - finalization 79
 - initialization 58
 - failure 59
 - life cycle 51
 - life cycle events
 - handling 52
 - service methods 59
 - notifying 81
 - programming long running 82
 - tracking service requests 80
- sessions 76
 - associating attributes 76
 - associating with user 78
 - invalidating 78
 - notifying objects associated with 77
- setContent method 316
- setNamespaceAware method 247
- setPostalAddressess method 371
- Simple API for XML Parsing
 - See SAX
- Simple Object Access Protocol
 - See SOAP
- SingleThreadModel 56

- SMIL 197
 - SOAP 250, 251, 269, 286, 290
 - body 307
 - adding content 313
 - envelope 307
 - faults 319
 - part 257
 - populating 260
 - specification 250
 - SOAP with Attachments API for Java
 - See* SAAJ
 - SOAPBody object 257, 293, 305, 307, 313
 - Content-Type header 316
 - SOAPBodyElement object 305, 307, 313, 332
 - SOAPConnection object 258, 296, 308
 - getting 302
 - SOAPConnection.call method 339
 - SOAPElement object 306, 332
 - SOAPEnvelope class 293
 - SOAPEnvelope interface 307
 - SOAPFactory class 305
 - SOAPFault object 319, 336
 - creating and populating 321
 - elements
 - Detail object 320
 - fault code 320
 - fault string 320
 - SOAPFaultt object
 - fault actor 320
 - SOAPFaultTest 336
 - running 338
 - SOAPHeader object 293, 305, 312
 - SOAPHeaderElement object 312
 - SOAPMessage class 293
 - SOAPMessage object 257, 304, 311
 - SOAPMessage.getAttachments method 318
 - SOAPPart class 293
 - SOAPPart object 294, 306, 314
 - adding content 314
 - SOAPPart.setContent method 314
 - SOAP-RP profile 299
 - SOX 194
 - specifications 181
 - SQL viii, 18
 - standalone 181
 - applications 324
 - client 301, 308
 - See also* client
 - standalone client
 - See* client, standalone
 - static stubs 277
 - stubs 277
 - stylesheet 183
 - SVG 197
 - system properties
 - com.sun.xml.registry.user-TaxonomyFileNames 369
- T**
- tag handlers
 - life cycle 146
 - tag library descriptors 122
 - filenames 118
 - listener 123
 - mapping name to location 118
 - tag 124
 - TagExtraInfo 129
 - taglib 118
 - tags 177, 179

- closing 179
- empty 179
- nesting 179
- taxonomies
 - finding with JAXR 363
 - ISO 3166 357
 - NAICS 357, 377
 - UNSPSC 357
 - user-defined 366
 - using to find organizations 358
- The Java Message Service Tutorial* 11
- transactions
 - Web components 58
- TREX 193

U

- UBL 199
- UDDI 263, 266, 327
 - getting access to public registries 351
 - registries 252, 348
- UDDI Server Registry 349
 - starting 375
 - stopping 382
- UnavailableException 59
- Universal Description, Discovery and Integration registry
 - See* UDDI registry
- Universal Standard Products and Services Classification
 - See* UNSPSC
- unparsed entity 222
- UNSPSC 357

V

- value types 272
- version 181

W

- W3C 193, 241, 269, 286
- WAR files 254
 - JavaBeans components in 28
- Web clients 7
 - maintaining state across requests 76
- web clients 25
 - configuring 26
 - internationalizing 38
 - J2EE Blueprints 39
 - running 34
 - updating 35
- Web components 9
 - accessing databases from 57
 - applets 9
 - concurrent access to shared resources 56
 - development role 16
 - forwarding to other Web components 74
 - including other Web resources 72
 - invoking other Web resources 72
 - mapping filters to 70
 - scope objects 55
 - sharing information 54
 - transactions 58
 - types 9
 - utility classes 9
 - Web context 75
- web components 25

- accessing databases from 39
- JSP pages
 - See JSP pages 26
- servlets
 - See servlets
- Web containers 13
 - loading and initializing serv-
lets 51
- Web module 28
- Web resources
 - mapping filters to 70, 71, 72
- web resources 28
- Web services 239, 250, 251
 - creating 253
 - discovering 266
 - RPC-based 250
 - writing a client application 254
- Web Services Description Lan-
guage
 - See WSDL
- web.xml file 405
- well-formed 203, 204
- World Wide Web Consortium
 - See W3C
- WSDL 251, 266, 270, 287
 - documents 252
 - using to find organizations
358, 377
- elements 292
- linking 194
- prolog 181
- registries
 - establishing security cre-
dentials 361
- transforming a DOM tree to
248
- XML Base 195
- XML data 216
- XML Schema 193
- XML Stylesheet Language Trans-
formations
 - See XSLT
- XPATH 191
- XPointer 195
- XSL 191, 247
- XSLT 191, 242, 247
- XTM 196

X

- XHTML 195, 202
- XLink 194
- XML 20, 177, 240, 269, 270
 - comments 180
 - content 181
 - designing a data structure 233
 - documents 216, 292